Interface

2003 November

［表紙デザイン：㈱プランニング・ロケッツ］

Contents 2003 November

特集

キャッシュ/MMU/例外/命令セットの詳細

＊

＊

＊：写真提供　志田晃一郎氏

Interface

Contents 2003 November

話題のテクノロジ解説

ショウレポート&コラム

一般解説&連載

■情報のページ

連載「XScaleプロセッサ徹底活用研究」「TOPPERSで学ぶRTOS技術」,「開発技術者のためのアセンブラ入門」,「音楽配信技術の最新動向」,「やり直しのための信号数学」は,お休みさせていただきます.

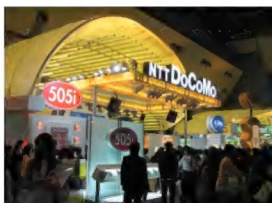
日本最大のワイヤレス専門展

WIRELESS JAPAN 2003

北村俊之

モバイルとワイヤレスの専門展示会「WIRELESS JAPAN 2003」が，7月16日(水)～18日(金)の3日間，東京ビッグサイトで開催された．主催は(株)リックテレコム．「ワイヤレスで考えワイヤレスで創造する 情報社会の近未来がここにある」をテーマに，約170社が出展した．「ユビキタス社会」をキーワードに，携帯電話とアプリケーションを組み合わせた最新利用方法，無線LANを利用した企業ネットワーク，ITSなど自動車による情報通信ネットワークなどが来場者の関心を集めていた．また，同時開催の「WIRELESS CONFERENCE 2003」では，NTTドコモ，KDDI，J-フォン3社の代表による，3Gサービスなどの今後のビジネス展望や経営戦略を披露するなど，こちらも盛況であった．

● 人気の焦点は携帯電話

展示会場でもっとも来場者の関心が高かったのは，何といっても携帯電話通信事業社であるNTTドコモ，J-フォン，KDDI，DDIポケットの4社と，それら事業者向けに端末を提供するメーカーのブースだった．NTTドコモ(**写真1**)のブースでは，「FOMA」とｉモード対応携帯電話「505i」シリーズの2本立てで展示を行っていた．505iについては，未発売のパナソニック モバイルコミュニケーションズの「P505i」の展示も行われ，FOMAでは，7月18日に発売された「FOMA N2102V」が展示されていた．

〔**写真1**〕**NTTドコモのブース**

J-フォンでは，10月1日の「ボーダフォン」へのブランド変更を控え，ブース正面に「vodafone」の巨大なロゴを表示するなど，ボーダフォンブランドを強く印象づけるブース構成となっていた．プレゼンテーションも，海外ローミングなどグローバルサービスを展開するボーダフォングループの特徴をアピールするものとなっていた．

KDDIでは，業種/業務別にビジネスアプリケーションを提供する「KDDIモバイルソリューション」，ECサイトや決済システムを構築する「KDDI ECソリューションサービス」，配車/運行管理やルートセールスなどを支援する位置情報提供サービス「KDDI GPS MAP」などの各種ソリューションをパネルやデモンストレーションによって紹介していた．また，PDAサイズのモバイルTV(**写真2**)の展示も行われており，来場者の注目を集めていた．DDIポケットでは，最大128kbpsのパケット接続を，倍の256kbpsに高速化する技術「AirH"」のデモを行っており，来場者の関心を集めていた．この技術では，1本あたり64kbps(従来は32kbps)のパケット通信網を4本束ねることで，最大256kbpsの高速通信を可能にしているとのことだった．ただし，あくまでも実験レベルで製品化にはまだ時間がかかり，実際のサービス開始時期などについては未定とのことだ．

〔**写真2**〕**KDDIのモバイルTV**

各通信事業者に対して端末を提供する，NECやパナソニック モバイルコミュニケーションズ，東芝，京セラ，三洋電機などのメーカーも多く出展していた．三洋電機では，第3世代対応端末のコンセプトモデルが紹介されていた(**写真3**)．こうした中でちょっと変わったｉモード端末を展示していたのが，日本無線である．同社の「ムーバR692i」(**写真4**)は，世界初の水に浮く，耐水ｉモード携帯とのことであり，実際に同製品を水槽に浮かべたデモを行っていた．24時間程度なら，水に浮いていても大丈夫とのことであった．

〔**写真3**〕**三洋電機のコンセプトモデル**

〔**写真4**〕**水槽に入れられた日本無線のムーバR692i**

● 周辺のテクノロジ

日本ノーベルは，携帯電話の新機能検証に対応した，「携帯電話自動評価システム」の新バージョン3.7の展示を行っていた．新バージョンでは，背面液晶の撮影と判定，QVGAレベルの高解像度画像の判定，2台のシステムによる対向試験などの新機能を装備している．また，RFテスタや基地局シミュレータなどの外部機器との接続もサポートしている．

ディアイティでは，IEEE802.11gをサポートする無線LAN製品群「Proxim ORINOCO」ファミリ，および無線LAN管理ツール「Wild Packets AiroPeek」シリーズの展示を行っていた(**写真5**)．富士写真フイルムでは，カメラ付き携帯電話用ディジタルプリンタ「NP-1」(仮称)の展示デモが行われていた(**写真6**)．同製品は117.5×41.5×105.5mmという小型サイズで，赤外線通信機能付きのカメラ携帯から信号を読み取り，約2分でインスタントカラーフィルムに印刷できる．1度に10枚のフィルムを収納することができ，バッテリ部はリチウム電池CR2を2本を使用．製品化は年内の製品化をめざしているとのことであった．

〔**写真5**〕**ディアイティのAiroPeekシリーズ無線LAN監視ソリューションRFGrabber**

〔**写真6**〕**富士写真フイルムのNP-1(仮称)**

コーンズは，無線通信における研究開発をサポートする製品を多数展示していた．構内無線システム設計ツールは，WLANや3Gの屋内アンテナの設置場所のシミュレーションが可能．また，Bluetoothプロトコルアナライザや認証テスタ，MPEG-4対応高速低消費電力画像処理デバイスなどの展示が行われていた．

オープンプラットホームによるテレマティクスサービスを実現しているモバイルキャストは，次世代カーテレマティクスとして，米国On2テクノロジーズのVP6コーデック技術を使用した，車のシートに設置された液晶画面にムービーを映し出すデモを行っていた．ルネサス テクノロジでは，携帯電話用マルチメディアアプリケーションの高機能化を実現する，携帯電話用アプリケーションプロセッサ「SH-Mobile」，および同プロセッサを搭載した携帯電話を前面に押し出したデモを行っていた(**写真7**)．

〔**写真7**〕**SH-Mobileを採用した携帯電話の数々**

ハッカーの常識的見聞録

35

広畑由紀夫

今月の常識

「端末のセキュリティを高めよう！」

☆ 今回は，この夏のBlaster/Nachi/Gaobotウィルス被害結果から，あらためてウィルス対策について考えたいと思います．

● Blaster と Nachi

Blasterウィルスは，亜種も含めてマイクロソフト社より公開されている「MS03-026」情報などの脆弱性を悪用して感染し，8月末にいたってもまだ被害を与えているワームです．脆弱性公開から対策用パッチまでそろった後であったことと，WindowsUpdateなどにより自動的にパッチをインストールしているコンピュータが多かったためか，サーバとして動作するコンピュータへの被害は，CodeRedのときに比べてはるかに少なかったと思います．

さらに，初期の被害報告が出てからBlasterウィルス自身の亜種が広がるまでに，初期のBlasterウィルスのバグのために対策を講じる余裕があったことなどからも，管理者が他者の被害状況を見て対策の必要性の確認と対策自身を開始できたこともあるかと思います．

Nachiは，基本的にBlasterの亜種といえるワームですが，Blasterウィルスと異なりICMPパケットなどを過剰に放出することや，TFTPサービスをインストールするなど，その動作において駆除すべきウィルスであることは同様だと思います．

● Gaobot にみる今後の対策

GaobotはNachi以降に現れたBlasterウィルスの亜種で，より広範囲に広がるべくIDやパスワードの辞書組み合わせによるファイル共有へのアクセスも試み，外部からの攻撃用ポートを開くことも行います．そのため，悪用される特定ポート番号が未使用であるならば，個別にそのポートを閉じておくことで同様の攻撃を防ぐことは可能です．

ただし，Windows XPにおいて，TCP/IPに対するファイアウォール設定で，IPアドレスについては全般的にセキュリティをかけられますが，個別ポートに対する設定までは，現状では市販セキュリティソフトに頼ることになるでしょう．

● 筆者のまわりの被害報告と対策

今回のBlasterウィルスについて，亜種も含め筆者のまわりで感染した多くのコンピュータは，一般的なコンピュータ販売店で購入後，WindowsUpdateなどをまったく使用せずに更新を怠っていたため，被害に遭ったようです．これら一連のウィルスによる被害は，筆者のまわりでは幸いなことに少なく，むしろ今回の騒ぎを見て大騒ぎして問い合わせてくる，今回の脆弱性とは無関係なトラブルの対処に筆者は追われていました．

筆者の取っているウィルス対策は，次のようなものです．家庭内LANを使用して外部と接続しているコンピュータに対しても，その多くは適応できると思います．実際，筆者の家庭内LANも，基本的にこのような対策をしてウィルスなどの攻撃から防いでいます．

① 外部インターネット接続箇所には必ずファイアウォールを入れて内部を保護する
② WindowsUpdateなどのOSパッチは，端末にはできるだけ速やかに自動的にインストールする
③ ウィルス対策ソフトをインストールし，ネットワーク共有フォルダ内も検査対象に含め常時監視する
④ サーバとして使用しているため，自動的にアップデートしたくない場合は，ダウンロードだけでもバックグラウンドで行っておき，システムが壊れる可能性と，ウィルスなどの攻撃によるシステムダウンの可能性などを考慮したうえで，パッチのインストールに関してスケジュールを作成し進める
⑤ 会社に接続される個人のコンピュータは，管理者より与える認証なしでネットワーク内に接続させない．また，物理的に接続しても他PCと共有できないよう，相互認証をとるように設計する
⑥ 無線LANに関してはMACアドレスも含め，登録コンピュータ以外は利用できないよう制限する
⑦ 外部と接するサーバはIP/プロトコル/ポート単位で制限をかけることの可能なファイアウォールをウィルス対策製品とともに使用し，必要のないポートは閉じるようにする

これらを行うだけで，被害の多くは防げると思います．また，被害にあったPCが持ち込まれても，そのウィルスの繁殖を可能なかぎり安いコストで防げると思います．しばらく前までは感染したコンピュータのソフトウェアやデータ破壊がおもなものでしたが，昨今のウィルスは，他の目的を併用するウィルスが増えてきたようです．感染しても自分自身の直接的に目に見える被害が少なくなってきたためか，対策をとらない個人ユーザーが増えてきたようにも思えます．そうであるからこそ，今後はよりいっそう事前の対策を取っておくべきだと思います．

- **マイクロソフト社Blasterウィルス対策情報サイト**
  http://www.microsoft.com/japan/technet/treeview/default.asp?url=/japan/technet/security/virus/blasterE_xp.asp
- **Symantec社 Norton AntiVirus2004 インフォメーションサイト（英語）**
  http://www.symantec.com/nav/nav_pro/features.html

ひろはた・ゆきお　OpenLab.

移り気な情報工学 35

ビットの化石

山本　強

長い間，筆者の研究室の棚には1枚の8インチ・フロッピーディスクが置かれていた．その8インチFDの物理フォーマットはいわゆる片面単密度なので最大240Kバイトしか入らない．それにはタイプで打ったラベルが貼ってあり，CP/M 1.4と書かれてある．そのエンベロープの中には登録用の返信用はがきがはさんであるが，そのあて先はパシフィックグローブのデジタルリサーチ本社になっている．

これは1978年当時のCP/M 1.4版のマスタディスクなのである．しかも，デジタルリサーチ社から直接購入したもので，個人的に日本に持ち込んだ正真正銘のマスタディスクという逸品で，今なら「お宝鑑定団」で値がついても不思議ではない（?）代物である．今年は2003年だから，1978年から数えると25年目という切りのよい年でもあるし，このディスクの中を読み取ってディジタル保存してみようと，ふと思い立った．

今ならまだ8インチFDのデータを読み取るFDドライブも残っていると思うが，いつまでもあるという保障はないだろう．そこで，歴史を残すという意味でも，今のうちに中のデータをビットとして再生しておくことは意味があるはずである．

ビットの発掘

さて，ディスクからデータを読もうと思っても，残念ながら8インチFDドライブは筆者の研究室にはない．もちろんCP/Mが動くマシンもない．幸い，各種メディアを新しいフォーマットに変換してくれるサービスがあるということを聞き，物理イメージのバックアップとCP/MファイルのWindowsファイルへの変換を発注してみた．何週間かたってビットデータが納品されたのだが，そのファイルを見てびっくりする発見があった．

筆者は，このFDの中はCP/M 1.4のマスタディスクのオリジナルイメージだけが入っていると思っていたのだが，中にはPIPやSTATといったトランジェントコマンドだけではなく，1980年当時のマイクロソフト社のCP/M向け開発環境一式が入っていた．マクロアセンブラM80やフォートランコンパイラF80，さらにはマイクロソフトの出世作であるMBASIC5.0版や筆者が若い頃にお世話になったBASICコンパイラBASCOMが，おそらく動くと思われる状態で残っていたのである．

CP/M時代のプログラムといえば関連ファイルは何もなく，.COMのファイル1個あれば動くものだったのである．今では，大切なマスタディスクをワークディスクのように上書きするとは非常識だと思うのだが，この非常識さが幸いして歴史的なビットデータがまとまって筆者の手元に残ることになった．世の中，何が幸いするかわからない．

エンジニア的な視点として興味深いのは，各プログラムのサイズである．このディスクは片面単密度で，全体で240Kバイトしか入らないのだが，その中にCP/M 1.4本体に始まってM80，L80，F80，MBASIC，OBASIC，BASCOMまですべて入ってしまうのが昔のプログラムだったのである．

ビットの再生

CP/M時代の「ビットの化石」をWindowsファイルに変換するという発掘作業が終わったので，次はその中身を調べるという作業に入った．まず手始めに，.COMファイルの中に埋め込まれているテキスト部分を抜き出して，そのバイナリファイルが作成された日付けや開発時代の痕跡を捜す．そのためには難しいことをする必要はなく，単にテキストエディタでバイナリファイルを開くだけで，中にあるASCII文字だけがそれらしく見えてくるはずである．

筆者が期待したのは，当時（1980年頃）ならMBASICの開発ではビル・ゲイツ氏もコンソールに向かってプログラムを書いていたはずだから，その名前がどこかに残っているのではないかということだったが，出てくるのは単なる日付とバージョン情報，それと数少ないエラーメッセージ程度だった．FD1枚の容量が240Kバイトという時代なので，無駄なデータは少しでも削るというのがその頃の常識だったのかもしれない．

次に行ったのは，このバイナリデータを実際に動かすことである．このディスクはマスタディスクなので，8080のエミュレータに簡単なBIOSをつけると自らブートできるのだが，ネット上にはCP/Mエミュレータというのもあるらしいので探してみた．そこでまず，簡単なほうからやってみた．CP/Mエミュレータをダウンロードしてコマンド引き数に`MBASIC.COM`をつけて立ち上げると，25年前のMBASICのプロンプトが再現されてしまった．当たり前だが，ビットの化石は劣化しない．

ビットの発掘は続く

誌面の都合もあるので詳細は次回ということにするが，この発掘作業は目下継続中であり，近日中にも25年前の開発環境がPC上に再現されることになっている．その顛末は追って紹介することにしたい．

ちなみに，このビットの化石状態のマスタディスク，実物はこういうものである．

```
http://nis-ei.eng.hokudai.ac.jp/~yamamoto/
                                otakara.html
```

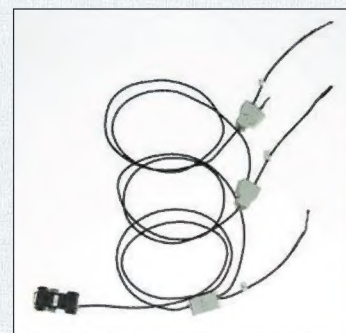
やまもと・つよし　北海道大学大学院工学研究科電子情報工学専攻
計算機情報通信工学講座　超集積計算システム工学分野

1線式デバイスによるWebベース多点温度計測

従来は多くの測定箇所で温度を測定しようとすると，かなり高額な測定機器が必要でした．しかし温度センサDS18B20(Maxim)の登場により，このような環境が1,000円程度のセンサを1本の線につなぐだけで実現できるようになりました．

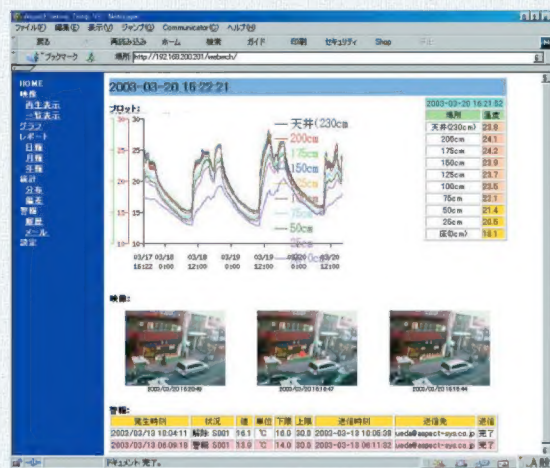
そこで今回は，データベースを使った多点温度計測システムを，Linuxマシンを用いて手軽にWebベースで実現します．

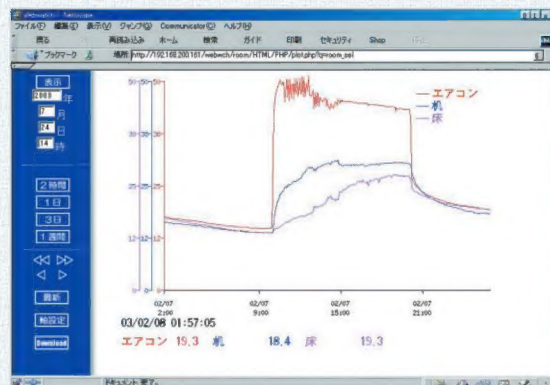
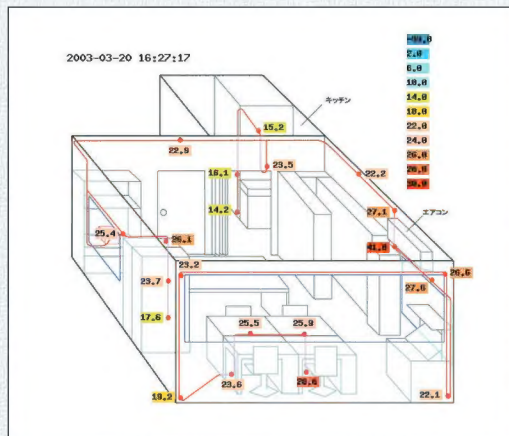
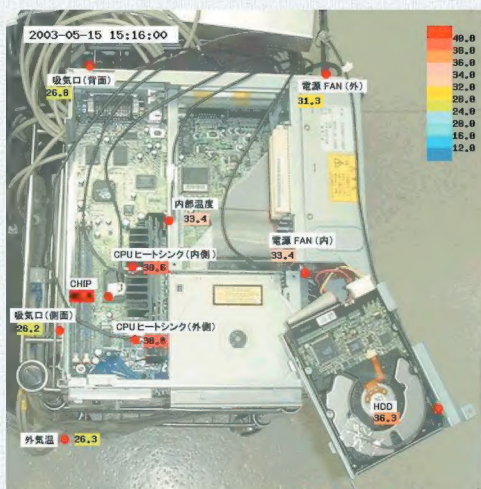
本文はp.127から始まります．

図A　温度センサDS18B20
今回使用する1線式温度計測デバイス．ユニークな64ビットIDをもち，写真のデバイスは0x287DC31D00000005のIDをもつ．DS18B20のセンサは先端が5mm四方程度の大きさ

図B　3点の温度取得機器全体像

図E　製品版計測システム

図C　取得機器
左から分岐コネクタ/ケーブル/センサ(DS18B20)/ショートピン

図D　RS-232-Cアダプタ(DS9097U)
DS18B20をPC/AT機などのシリアル端子に接続できる

図F　Web統合画面

図G　オフィス各点の温度
冬の1日でオフィス全体に張り巡らした各センサからの値は，エアコンを入れてゆっくり温まっているようすを表している

図H　冬の1日のオフィス温度変化
1日のプロットでいちばん上がエアコン出口，下が机と床の温度

図I　パソコン内各部の温度
パソコンの各部を測ってみると，電源を投入してから各部の温度が上がるようすや，CPU負荷の大きいプログラムを実行するとCPU温度が上がることがわかる．また待機電力も思いのほか大きく，電源を切ることで省エネにつながることもわかる

特集 キャッシュ/MMU/例外/命令セットの詳細

マイクロプロセッサ技術の基本

Contents

特集執筆：中森　章

今月号は，2号連続のマイクロプロセッサ解説特集の第2弾である．

MPUを高速で動作させるためには，キャッシュシステムは必須のものとなっている．第1章では，現在のMPUで使われている各種キャッシュの構造やその動作について，くわしく解説する．

WindowsやLinuxなどでは仮想記憶が使われている．仮想記憶を実現するにはMMU機能が必須である．第2章ではMMU機能の基礎から，x86系や680x0系などのMMUのしくみを解説する．

MPUの応用範囲を大きく広げる機構として，割り込みや例外が上げられる．第3章では，割り込みと例外，ハードウェア割り込みとソフトウェア割り込み，内的要因と外的要因など，割り込み動作の本質を解説する．

最後に，「究極のCISC」と呼ばれるMPUから，初期のRISCプロセッサ，現在のRISCプロセッサ，そして最新のMPUに取り込まれているSIMD系命令まで，これまで登場してきた各種MPUの命令セットアーキテクチャの変遷を振り返ってみる．

Prologue

研究段階から実用化へ，そして現在残っているのは……

RISCプロセッサ興亡史

中森 章

はじめに

● RISCはCISCに対するアンチテーゼ！

RISCはCISCに対するアンチテーゼとして開発された．その発想は，CISCではむだな命令が多く，それが動作周波数を上げる妨げになっているというものだ．**洗練された簡単な命令セットにすれば，パイプライン処理を効率的に動作させることができ，動作周波数の向上とあいまって最高の性能を達成できる．**この観点で，1980年代から，大学や企業でさかんにRISCの研究が行われるようになった．

CISCはある時期まで，思いつきでアーキテクチャを拡張していった感があるが，RISCは最初から定量的に性能評価をしながら論理的にアーキテクチャを決定していった．

● CISC……いや，x86の一人勝ち？

RISCは，本来は動作周波数を上げるための技術であるが，現時点ではx86系MPUのほうが動作周波数が高い．結局は，プロセス技術の開発に莫大な投資をできる企業だけが生き残ることができるのだ．投資回収という観点から見れば，需要が知れているサーバやEWS向けのRISCの利益が少ないのは明らかで，莫大な数が出荷されているPC用のMPUには太刀打ちできない．金持ちはより金持ちに，貧乏はより貧乏に，もしくは理論派が現場からの叩き上げに負けたようなものであろうか．

しかし，x86がRISCの技術を採り入れながら性能（とくにIPC）の向上を図ってきたのは事実であり，その上でプロセス技術によるさらなる高速化がなされたと考えることができる．その意味で，x86の進化にあって，**RISCは高速化のためのアイデアを提供してきた**ともいえる．Intelと並ぶ優秀なプロセス技術を有するIBMは，自社のRISCであるPowerPCで一人気を吐いているが，ほかのRISCはサーバ/EWS/PCなどの高性能が要求される分野からは早晩消滅していくかもしれない．

● RISCの進むべき道は

RISCの進むべき道は，性能よりも超低消費電力が要求される組み込み制御の分野であろう．CISCは性能を追求するあまり，消費電力には無頓着になっている．最近でこそIntelなども電力削減のスローガンを掲げているが，ARMやSHやMIPSなど，早い時期から低電力を売りにしてきたRISCには追いつけない．GHzを超える動作周波数を維持しようとする限り，永久に追いつくことは不可能であろう．そこにRISCの生きる道があると思われる．

RISCには，x86のようにPCを中心とした共通の応用分野があるわけではなく，各コンピュータメーカーが自社のサーバやEWSを高速化できればいいという発想で開発が続けられたので，そのアーキテクチャは各社バラバラである．したがって，その解説はオムニバス形式にならざるを得ない．

ARMやSHやMIPSは，組み込み制御分野を自分の色に塗り変えようと必死の努力をしているが，x86のように世界統一できる日が来るのだろうか．

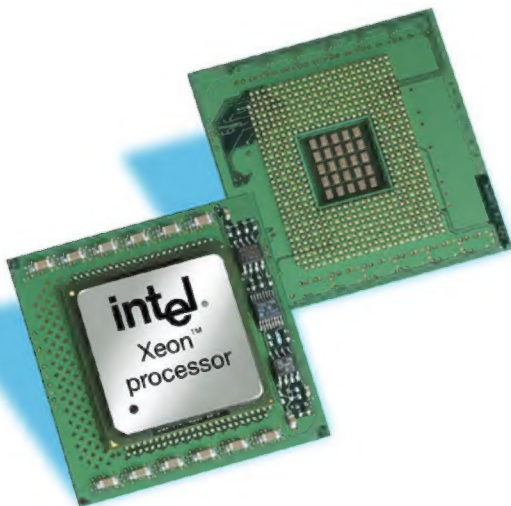
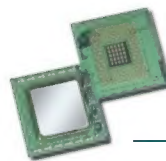
〔写真〕x86系の一人勝ち!?

RISCプロセッサの興亡

以降では，RISC系MPUの興亡史を簡単にまとめてみる．

● IBM801

1975年，IBMは高級言語のプログラムを用いて既存のものよりも非常にコスト/パフォーマンスのよいシステムを提供するため，ハードウェア，OS，コンパイラを含めたコンピュータシステムの設計に着手した．その設計思想は，ほとんどの命令の処理は1クロックで終了するべきというものだった．これは，コンパイラが生成する命令コードは基本命令（ロード，ストア，分岐，比較，加算）ばかりであり，それらを高速化すれば全体の性能が向上するという根拠に基づく．その研究の結果生まれたのがIBM801である．しかし，なぜかIBMは1982年までその存在を公表しなかったという．

IBM801の扱うデータ型はキャラクタ（バイト），ハーフワード，ワードの3種類で，アドレシングモードは「ベース＋インデックス＋オフセット」という単純なものだった．命令長は1ワード固定で，3オペランド演算を行った．また，当時は汎用レジスタの本数は16本が主流だったが，IBM801は32本のレジスタを備えていたのが特徴的である．IBM801はECLプロセスで製造され，決してVLSIと呼べる集積度ではなかったが，命令キャッシュなどRISCの特徴を備えており，後年のカリフォルニア大学バークレー校やスタンフォード大学のRISC研究に大きな影響を与えた．

そのわりには，IBM自身はRISCの有効性に気付いていたとは言い難く，IBMがRISCに本格的に手を染めるのは，1992年のPowerPCになってからである．

● バークレーRISC Ⅰ/RISC Ⅱ

カリフォルニア大学バークレー校のRISCの研究は，高級言語コンパイラが複雑な命令を有効に使えないという事実に着目することから始まった．プログラム実行時の命令の出現頻度，アドレッシングモード，変数の使われ方などの統計を採り，新しい命令セット設計の指針とした．この研究結果は，1980年に同大学のPattersonとDitzelによって初めてのRISCに関する論文『The Case for the Reduced Instruction Set Computer』として発表された．この論文は，シングルチップコンピュータに最適なアーキテクチャはRISCであると主張し，その理論をバークレー校の大学院生が実践したのがRISC ⅠとRISC Ⅱである．これらは当時のCISCよりも単純で，設計の労力も少なかったが，CISCに匹敵する性能を発揮していた．かくしてバークレーRISCは，後のARMやSPARCアーキテクチャの基礎となるのである．また，RISCという言葉はバークレー校によって初めて使用された．

ちなみにDitzel氏といえば，Crusoeを開発したTransmeta社の創業者としても有名である．RISCを提案した彼がCrusoeではVLIWを選択したところが興味深い．

● スタンフォードMIPS

バークレーRISCと同時期，スタンフォード大学でもHennessyを中心にRISCの研究が行われていた．それは以前のRISCとは毛色が異なり，ハードウェアをできるだけ単純にして，レジスタの依存関係等の煩雑な処理は「リオーガナイザ」と呼ばれる再構成ソフトウェアで解決しようと試みた．パイプラインを効率的に動作させるためにハードウェアによるインターロックを許さず，ソフトウェアで命令の順序を並び替えて正常動作を保証する．

このためスタンフォード大学のRISCは『Microprocessor without Interlocked Pipeline Stages（パイプラインステージがインターロックしないマイクロプロセッサ）』の頭文字を採ってMIPSと呼ばれた．もちろん，コンピュータの性能を示すMIPS（Million Instructions Per Second）との掛詞である．

MIPSではハードウェアを簡単にするために，メモリアクセスはワードアクセスのみとし，バイト単位の操作が必要な場合は専用命令を使ってレジスタ上で処理する．また，汎用レジスタは16本だった．

最初のMIPSプロセッサは実用的といえるものではなかった．しかし，スタンフォード大学の研究者たちは，その研究を推し進め，2Kバイトの内蔵命令キャッシュと256Kバイトの外付け混合キャッシュインターフェース，32本の汎用レジスタ，乗除算用の特殊レジスタ，ゼロレジスタ，5段パイプラインを特徴とするMIPS-Xというプロセッサを設計した．これは後のR2000とほぼ同じ構成であるが，分岐の遅延スロットが2命令である点が決定的に異なる．

● R2000以降のMIPS

スタンフォードMIPSの研究成果を受けて，1984年にベンチャー企業のMIPS Computer Systems（現MIPS Technologies Inc.）が生まれ，R2000をはじめとする非常に高速なプロセッサファミリを世に送り出すことになる．これは，Hennessy教授の研究成果を誰も信じてくれなかったため，RISCで高性能を達成できることを実証するために自らMIPS社を創設したものである．

1991年，MIPS社は世界初の64ビットMPUであるR4000を発表する．CPU，FPU，L1キャッシュ，L2キャッシュインターフェース，マルチプロセッサ機能を1チップに内蔵した画期的なものだった．

MIPSのMPUは，それまでおもにSGI（Silicon Graphics Inc.）のGWS（Graphics Workstation）に採用されていたが，SGIは1992年にMIPS社を買収して超高性能なMPUの開発を行わせることになる．それがR10000をはじめとするハイエンドプロセッサシリーズである．R10000の開発と並行して，MIPS社はローエンドのR4200/R4300も開発している．R4300はNintendo64に採用されることを目的として開発されたMPUである．

この頃になると，元MIPS社の開発メンバがスピンアウトしてQED（現PMC-Sierra），SandCraftといった新しい会社を設立し，MIPS社と独立にMIPSアーキテクチャのMPUを開発するようになった．

そして1998年，SGIが自社のGWSのMPUとして将来的にIntelのIA-64プロセッサ（Itanium）を採用することを決定すると，MIPS社はライセンス管理とIPコアの販売を目的としてSGIから分社化される．分社化されたMIPS社は，それまでのMIPSアーキテクチャを，32ビットのMIPS32，64ビットのMIPS64として再整理した．そしてそのほかにも，特定分野向けにMIPSアーキテクチャの拡張を積極的に行うようになった．たとえば，組み込み制御用に16ビット長の命令でコード効率を上げるMIPS16，3次元グラフィックス用のMIPS-3D，スマートカードやJava用のSmartMIPS，ネットワークアプリケーション用拡張などを提唱している．

現在では，SGI，MIPS社以外でも，独自にMIPSアーキテクチャのプロセッサを製造するようになっている．代表的なところでは，PMC-Sierra，IDT，NEC（V_Rシリーズ），東芝（TXシリーズ），Alchemy，SiByte（現在はBroadcomに買収された）がある．とくにAlchemyとSiByteは，DECでAlphaチップやStrongARMチップを開発していた技術者がスピンアウトして設立した会社である．いわば，他のアーキテクチャからの乗り換え組である．

2002年2月，Alchemyはx86互換メーカーの老舗であるAMDに買収された．AlchemyはAu1000/Au1500というMPUで，400MHzで500mW，500MHzで900mWと，高動作周波数にもかかわらず低消費電力を達成していることで定評がある．IntelがStrongARMやXScaleでインターネットアクセス系の組み込み分野に進出しようとしているのに対抗する形である．IPコアの分野において，MIPSはARMよりもやや不利な状況にあったが，この買収劇でMIPS陣営が活気付くかもしれない．さらに2002年4月29日，AMDはMIPS64のライセンスを取得した．AlchemyはMIPS32のライセンスしかもってなかったが，これでAMDは思う存分MIPSチップを製造できる．

2002年10月のMicroprocessorForumでは，Broadcomが1GHzのMIPS64コア（SB1）を1チップに4個内蔵するBCM1400を発表した．消費電力は，1GHz動作時に1チップ全体で25Wとかなり小さい．同フォーラムで東芝は，800MHz動作のTX99（Amethyst）をCPUコアとするSoCであるTMPR9961を発表した．CPUコアはMIPSでありながら，内部バスがARM系のAMBAバス（AHB）であることが興味深い．また同時期，NECはV_R5500（コードネームSapphire）というCPUコアで800MHz動作を達成したと発表し，2003年には1GHz版の開発を完了すると発表した．PowerPCやx86系のMPUに対して周波数の向上が遅れていると酷評されることが多いMIPSアーキテクチャであるが，着実に動作周波数を上げてきている．

現在，MIPS64系のIPコアは5Kのみとなり，20K（Ruby）と25Kf（Amethyst）が新しいロードマップからは消え失せている．これは，MIPS社がMIPS32アーキテクチャに注力しようという表れであろう．

新しいプロセッサの24Kという名称は明らかに25K(Amethyst)の下位チップであることを強調するものである．スーパースカラではなくなったが，シングルパイプで動作周波数を向上させていくことで20Kや25Kの置き換えをねらったものであろう．東芝はMIPS社と共同でTX99(25Kf, Amethyst)の開発を行っていたが，梯子を外された格好である．

これをもって，MIPS社はMIPSアーキテクチャを管理・維持していく求心力を失ったと見る向きもある．2003年5月にはSandCraft社も倒産しており，64ビットアーキテクチャ(MIPS64)を維持していくためには，NEC，東芝，MC-Sierra(QED)，AMD(Alchemy)，BroadCom(SiByte)の頑張りが期待される．しかし，MIPS社以外では，実際にフルスクラッチでCPUコアを開発しているのはAMDとBroadComとNECだけであり，MIPSアーキテクチャの衰退を伺わせる．

● ARM

ARMの歴史は，英BBC教育チャネルによる教育用コンピュータプロジェクトに端を発する．このプロジェクトがAcorn Computerに発展した．1983年，Acorn社は英国で成功を収めていた6502を搭載したBBCコンピュータ(一般にはBeepとして知られている)の次機種用のMPUを探していたが，市場に出ていたMPUで満足のいくものがなかったため，Acorn社はMPUを自社開発することにした．しかし，MPU設計の経験が浅かったAcorn社は，少ない設計労力で良い設計をする必要があった．そこに現われたのがバークレーのRISC Iの論文であり，それを参考に，約18か月を費やしてMPUが設計された．これがAcorn RISC Machine，つまりARMである．

ARMプロセッサは，当時PDA「Newton」を計画していたAppleの目にとまり，同機のCPUとして採用されることになった．これが契機となり，1990年にARMは，Apple，Acorn，VLSI Technologyの出資によって設立されたAdvanced RISC Machines社に移管された．そして，その3年後に「Newton」が発売された．

ARMはARM7でヒットを飛ばし，ARM8，ARM9とアーキテクチャを開発していく．

後年，ARM社はDEC(Digital Equipment Corporation)と共同でStrongARMを開発した．これは，高速動作を誇るDECのAlphaプロセッサの技術を応用して高い性能を引き出そうとするものである．ARMとしてはキャッシュ構成に初めてハーバードアーキテクチャを採用した．

一方，StrongARM部門はIntel社に買収された．正確には，1997年にDEC社がIntelをAlphaの知的所有権侵害として提訴した際の和解条件の一環として，DEC社が半導体工場をIntelに売却したとき，工場と一緒にARMの製造権利が付いてきた．IntelにすればPC用MPU以外の組み込み制御分野への事業展開を図るための手駒をおまけ同然(?)に入手したわけである．この状況に面白くないのはARM社である．ARM社自体はARM9，ARM10と独自の製品展開を行っていくが，StrongARMに関しては口を閉ざして何も語らなかった．

ARM社はARMの製造ライセンスを多くの半導体メーカーに与えているが，基本的に，ARMアーキテクチャの改造権はない(Motorolaには特別に改造権を与えているらしい?)．ARM社の提供する設計情報をそのまま使用しなければならない．これが，ICE機能を含め，確固たる互換性を実現している．

しかし，StrongARMを製造するIntelにはARM社の強制力はない．Intelは独自にStrongARMを改造して，より高速，より低消費電力なStrongARM2(後にXScaleとして発表された)を開発してしまった．しかしXScale登場当初は，バグが直らず散々だったようだ．

そして2001年7月30日，ARM社は次世代のプロセッサアーキテクチャ「v6」をIntelとTexas Instruments(TI)にライセンスすると発表した．ARMとIntelは不仲だったはずでは？　しかし，Palmが次期OS用のデバイスとしてARMアーキテクチャを採用するにあたり，Intel，TI，Motorolaと提携することを表明したのを契機に和解したのかもしれない(真相は不明)．XScaleの(バグや性能が低いという)評判を受けて，ARM社がXScaleを脅威に感じなくなったせいというのは穿ち過ぎか．

別の情報筋によれば，Intelはもともと ARMアーキテクチャをカスタマイズする権利を有していたという．今回契約を拡大し，「v6」だけでなく，ARM7，ARM9，ARM10のコアにアクセスする権利を得たという．ARMとIntelの不仲説は幻想だったのかもしれない．実際，いつの間にかXScaleの命令セットはARM v5TE互換ということになっている．これはXScaleも純粋なARMの系譜であることを強調するものである．現状，StrongARMとXScaleがARMのWebサイトに載っていることを思うと感慨深い．

2002年4月29日，ARMはEmbedded Processor ForumでARM11コアを発表した．これは，v6アーキテクチャに基づく最初の製品となり，2002年第4四半期にライセンスを開始するという．XScaleをARMの一員と認めつつも，ARM本家の意地にかけて，より高性能のMPUをぶつけてきたというところか．

● i860

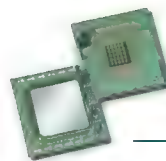
i860は1986年に開発が着手された．1988年の半導体技術を考えると，100万トランジスタを1チップに集積できるとし，x86とは異なる新しいアーキテクチャを提供する目的で開発された．結果，i386＋80387の性能を上回る高性能を実現することができ，従来のスーパーコンピュータやミニコンが提供していた科学技術計算や各種のシミュレーションをより小型で安価なシステムで実現できた．

i860/i960は，現在でもDSPの代用品やRAID用のプロセッサとして生き残っている．

● 88000

EWS用のMPUがRISCに傾き始めると，Motorolaもその風潮に迎合した．1988年から88000の出荷を開始した．当時のMotorolaの副社長は「混乱した32ビットプロセッサの競争で生き残ったのはIntelとMotorolaだ．RISCも同様だ」と自身満々のコメントを出していた．

88000とは，CPUであるMC88100とキャッシュとMMUを内蔵するMC88200というチップの総称である．コードユニット，データユニット，整数ユニット，FPU(加減乗除と変換用の二つ)の計五つのユニットが各自パイプラインで並行動作するという意味で，スーパースカラのはしりである．1988年に発表された．MC88100は比較結果を反映させる条件コードレジスタをもっていない．比較命令は，ほかの演算命令と同じく，3オペランド命令で，比較結果をデスティネーションレジスタに格納する．条件分岐命令はこのレジスタの値に基づいて分岐する．この構成により，比較命令と条件分岐命令間の命令を自由にスケジューリング(入れ替え)できる．条件コードを使用しないこの方式は，MIPSをはじめとする多くのRISCで採用されている．

88000の売れ行きは，最初は好調だったが，徐々に翳りを見せていった．88000の失敗の原因はいくつかあるが，最大の原因はIntelのi486対抗の68040と並行して開発をしたため，リソース不足となり次機種の開発が大幅に遅れてしまったことである．実際，88000と68040そのものも開発が遅れた．

● SPARC

SPARCとは『Scalable Processor ARChitecture（拡張性のあるアーキテクチャ）』を意味する．それ自体は命令セット，アドレシング，MMUのソフトウェアのみの規格であり，実装は半導体メーカーにまかされている．

現在のSPARCは，バークレーRISCの研究成果を基に1992年にSun Microsystemsが自社のEWSであるSUN4用のMPUのアーキテクチャとして提唱し開発したものである．その仕様はオープンアーキテクチャとしてSPARCインタナショナル社によって管理されている．そのメンバには Sunのほかに，富士通，TI（Texas Instruments），LSI Logic，Rossなどが所属し，各社独自にSPARCチップの開発を行っている．

SPARCにはいくつかのバージョンがあり，最新バージョンは9である．バージョン9は64ビットアーキテクチャだが，（少し前の）典型的なSPARCチップは32ビットアーキテクチャのバージョン7または8に基づいている．

2001年ごろから，Sun OS（Solaris）の64ビット化が急速に進行中で，MPUもバージョン9を実装するUltraSPARC IIIの750MHz品を搭載したマシンがSunの稼ぎ頭になってきた（なお，UltraSPARCシリーズの製造はTIが担当している）．またサーバやEWS分野だけでなく，組み込み向け分野では，低消費電力版SPARCとしてSPARCLite（富士通）などもシェアをもっている．

SunはIA-64のItaniumをライバルと考えている節がある．しかしIA-64同様，実際の製品出荷は発表したロードマップからは遅れぎみだ．Sunでは，現在，別々の開発チームがUltraSPARC IVとUltra SPARC Vの開発に従事している．UltraSPARC Vの動作周波数は1.5GHzを匂わせているが，1GHz動作のUltraSPARC IVも出荷されてない現状では，まだ海のものとも山のものともわからない．

● Am29000

AMD（Advanced Micro Devices）は，いまやIntelの最大のライバルであるx86互換MPUメーカーである．しかし，昔はIntelの8080A，8086，80386やZilogのZ8000のセカンドソースを製造する会社というイメージがあり，オリジナルなMPUには注力してなかった．

唯一，1978年に発表されたビットスライス型のAm2901ファミリがあった程度である．これらはコンピュータや専用制御回路を作成するためのLSI群だった．たとえば，Am2901は4ビットALUやレジスタを含むLSIであり，8ビットプロセッサでは2個，16ビットプロセッサでは4個を組み合わせて使用する．

そのAMDが，1987年に突如，オリジナルな1チップRISC型MPUを発表した．それが，バークレーRISCの流れを受け継ぐAm29000である．これは，同じバークレーRISCの流れを継ぐSPARCと同時期である．CISCでいえば，Intelの80386やMotorolaの68030と同世代である．

Am29000の特徴といえば，192個という大量の32ビット汎用レジスタである．これが，SPARCと同様のレジスタウィンドウ方式で使用され，64本がグローバルレジスタ，残りがスタックキャッシュとなる（SPARCのグローバルレジスタは8本）．命令体系は3オペランド形式で，分岐遅延スロットをもつ．この時期のRISCと同様に，乗除算命令はもっていないが，乗除算処理の1ステップ分を実行する命令が用意されている．これもSPARCとよく似ている．Am29000は4ステージパイプラインのスーパースカラ構造を採る．また，浮動小数点演算を行うためにはコプロセッサのAm29027が必要である．

Am29000の命令セットの特徴として，この時期のRISCには珍しく，CISCと同様なビット操作命令とビットフィールド命令をもつ．

Am29000は，EWSというよりも組み込み制御向けに使われたので，同時期のRISC系MPUの中では出荷数が多い．しかし組み込み分野でも，徐々にMIPSやSPARC系のMPUに押されはじめたAm29000は，1995年，突如「終了宣言」が出された．登場も突如だったが，終焉も突如だった．

以後，AMDはAm29000の技術を応用してK5を開発し，より利益の見込めるx86互換ビジネスへ注力するようになる．

● Transputer

かつて一世を風靡したTransputerは，英国INMOS社が設計した，マルチプロセッサシステムの構築を容易にするMPUである．INMOSは，1978年に英国の国策会社として設立された．設立初期はRAMを開発していたが，1980年に最初のTransputerを開発した．Transputerは，TransistorとComputerを併せた造語である．INMOSは1984年に民営化されている．1987年にはSGS-Thomson Microelectronics（現在のST Microelectronics）に買収されている．

INMOSは，1978年にOxford大学のCharles Antony Hoare博士によって提唱されたCommunicating Sequential Processes（CSP）理論に基づいて，Occamというコンピュータ言語を開発した．Transputerは，このOccamのプログラムを実行させるために開発され，その後の並列コンピュータの実用化に多大な影響を与えた．

Transputerは単純さを第一として開発された．たとえば，レジスタは3個しかなく（このほかに，FPUに3個，命令ポインタ，ワークスペースポインタ，オペランドレジスタの計9本がある），命令セットも単純である．内部的にはスタックアークテクチャに基づいたRISCである．

Transputerは最大4チャネルの双方向シリアルリンクをもち，これをほかのTransputerのシリアルリンクと上下左右に結合することで，並列的なネットワークシステムを構築できる．ネットワーク形態には，相互通信，パイプライン，ツリー，2次元格子，ハイパーキューブなどがある．このシリアルリンクとチップ内のメモリはDMAで通信される．

Transputerの最上位版であるT9000は1991年に発表された．T9000は，最高50MHzで動作し，8命令を同時実行するスーパースカラである．一説によると，このT9000の開発がうまくいかなかったため，Transputerのラインが途絶えてしまったという．

Transputer自体は簡単に並列システムが構成されるために，大学などの研究機関で多用されていた．しかし，通常のMPUでマルチプロセッサ構成が常識になっている現在では，その存在を知る人は少ない．

● 60xからG4までのPowerPC

PowerPCとは，IBMが大型計算機用に設計したPOWER（Performance Optimized With Enhanced RISC）アーキテクチャを1チップで実現するものである．本来はIBMのEWSであるRISC System/6000（RS/6000）のアーキテクチャを基にしている．RS/6000は科学計算を前提として設計されたが，PowerPCはノートPC，組み込みコントローラ，高性能科学計算用およびGWS（Graphics Work Station），マルチプロセッサ構成のメインフレームなどに対象を拡げた．

RS/6000は7～9の複数チップから構成されたが，PowerPCでは1チップの実装に適するように命令の削除，追加が行われている．とくに使用頻度が低く，IPCを上げにくい命令は削除または簡潔化されて

いる．追加命令は整数や浮動小数点の演算系に多い．

PowerPCは発表当時，RISCの中でも豊富な命令を備えた「Rich RISC」と呼ばれ，(今ではありふれているが)積和命令やレジスタ値に依存した分岐命令，OS専用命令が注目を浴びた．後々高性能化の妨げになるので遅延分岐は採用しないといったのは有名である．

1991年7月にApple，IBM，MotorolaがPowerPCに関して提携し，IBMとMotorolaが製造するMPUをIBMやAppleが自社のPCに採用することを決めた．IBMとAppleは，IntelとMicrosoftの寡占状態にあるPC分野において，新しいPCだけでなく情報機器の世界標準を作ろうと画策して，Motorolaを巻き込んだ形で提携を成立させた．Apple，IBM，Motorolaの連合は各社の頭文字を取ってAIM連合(後にはPowerPC連合)とよばれるようになった．AIM連合の成立から約1年後，1992年の6月にはテキサス州のオースチンにSomersetという研究所が設立されて本格的な開発が始まった．

1992年12月，IBMはすでに最初のPowerPCアーキテクチャのMPUとして，PowerPC601を開発していた．Somersetの最初の仕事はPowerPC601のシュリンク版の開発である．それは1994年に登場した．IBMの公式資料によれば，まず，POWERを1チップ化したRSCというMPUがあり，それを基にPowerPC601が開発されたとある．このあたりの事情は不透明である．その後，IBMとMotorolaは，PowerPC601に引き続いて，第2世代(G2)のPowerPC603/604/620をを開発することになる．

当初，Somersetでの製品開発は非常に順調に行っているように見えた．PowerPC連合は次期PCの標準仕様としてPReP(PowerPC Reference Platform)やCHRP(Common Hardware Reference Platform)という規格を制定して，PowerPCの市場への浸透を図った．1996年にはPC向けにPowerPC603eとPowerPC604e(604の1次キャッシュ倍増版)が発表され，Macintoshの68000系からPowerPCへの完全移行も発表された．そして，1997年に開発コードネームArthurと呼ばれていたPowerPC750(G3)が登場すると，MacintoshもPower PC603/604からPowerPC750に移行していった．

しかし，PowerPC連合の蜜月時代は長くは続かなかった．AppleがIBMの意向を無視してPowerPCをApple(Macintosh)に特化しすぎたため，IBMはMotorolaとの共同開発から撤退してそれぞれ独自のPowerPC路線を構築することになる．具体的には，性能向上のためのAltiVecの仕様をめぐる意見の対立が原因といわれている．また，MicrosoftがWindows NTでのPowerPCのサポートを途中で放棄したことも一因であろう．

その後，AppleとMotorolaは，1999年8月に，第4世代に相当するPowerPC G4(MPC7400)を発表した．これはPowerPC740/750に，結局はMotorolaが開発した，マルチメディア用SIMD命令であるAlti Vecを実装した製品である．G3，G4はMotorolaが開発したPower PC603をコアとした派生品であり，どれも最大2命令(＋AltiVec)同時発行，最大3命令同時実行のスーパースカラということになっている．

● Power4とPowePC970

IBMにはPowerPCとは別のロードマップとしてサーバ向けのラインナップとして，Power1，Power2，Power3がある．これらはPower PCを，より性能が出るように，SMP(Symmetric Multi-processor System)対応に改造したものである(Power1はPowerPC601と同一という話もあるが)．IBMの大型計算機であるRS/6000シリーズの開発チームによって独自に開発された．しかし，これらのチップとそれを搭載したサーバは，動作周波数が他社に比べて劣るため，市場でのシェアを徐々に失っていった．

Power4はそのような状況を打破するために開発された．Power4は，命令セット以外は，それ以前のPowerチップとは別物である．高速クロック技術は637MHzを実現したPowerPCの開発チームが担当した．高速バス技術，パッケージ技術も，それぞれIBM内の専門家チームが担当した．システム設計はRS/6000チームが担当した．CPUコアはPower3の技術者とSomersetの残留組が担当した．このように，その時点でのIBMがもてる最高技術を注ぎ込んで開発された．

2002年10月14日，Microprocessor Forumに先立ち，Power4を基にしたPowerPCであるPowerPC970が正式に発表された．1.8GHzで動作し，32ビットだけでなく64ビットアプリケーションの実行が可能である．さらにSMPをサポートする．Power4との最大の違いはCPUコアを一つしか内蔵しないこと，CPUコアの動作周波数が1.8GHzに引き上げられたことである(Power4は1.3GHz)．ベクタ命令に関しては，プレスリリースでは，単にSIMD命令をサポートするとなっている．

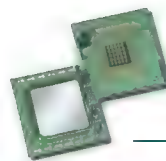
そして2003年6月23日，AppleはPowerMac G5を世界最高速で世界初の64ビットPCとして発表した．プレスリリースによれば，CPU(PowerPC G5)はAppleとIBMの共同開発となっているので，PowerPC970と考えて間違いない．G4のAltiVecをめぐって一度は袂を分かったAppleとIBMだが，なかなかクロックの上がらないMotorolaのG4にしびれを切らせたAppleが，IBMとよりを戻したといったところか．

● PA-RISC

PA-RISC(Precision Architecture RISC)とはHP社のEWSであるHP9000シリーズのアーキテクチャであり，EWSの分野ではかなりの実績をもつ．それでいて，ビット操作命令，ビットフィールド命令，独自機能をサポートするSFU(Special Function Unit)を有し，組み込み制御分野にも適している．事実，1991年に発表されたHD69010(PA/10)は，EWSだけでなく組み込み分野の応用も見込んでいた．

PA-RISCのアーキテクチャは，HPと契約を結んだメーカーにしか公表されていないので詳細は長らく不明だった(現在は，その概要をHPのサイトで知ることができる)．ただ，初期のPA-RISCではメモリアクセスを非常に高速にし，その代わりL2キャッシュを使用しないことを公言していた．メモリアクセスが十分高速なら容量に制限のあるL2キャッシュは意味がないというわけである．この思想が現在も生きているか否かは不明だが，メモリアクセスを高速化したければ，DRAMでなく(高価でも)SRAMを使えという主張は核心を突いているかもしれない．

1998年に発表されたPA-8500では，巨大な4ウェイのL1キャッシュ(命令0.5Mバイト，データ1Mバイト)を内蔵している．2001年に発表されたPA-8700に至っては，命令0.75Mバイト，データ1.5Mバイトと超巨大なL1キャッシュを内蔵する．これはチップ面積の3/4以上を占め，MPUではなくSRAMチップと揶揄する声もあるとか．ただし，相変わらず，L2キャッシュはサポートしない．PA-8700は800MHz以上で動作する．

HP自体はEWS用のMPUをIntelのIA-64に移行することを表明しているので，PA-RISCが幻のアーキテクチャのまま終わってしまう可能性は大きい．とはいえ，IA-64の開発遅れに危機感をもっているのはHPも同様らしい．2000年に発表されたロードマップでは，2001年に800MHz動作のPA-8700，2002年に1GHz動作のPA-8800，そして

将来的に1.2GHz動作のPA-8900の開発を行うことが明示されている．PA-8900以降は完全にIA-64に移行する予定であるが，これはItanium，Mckinleyに続く第3世代であるDeerfieldやMadisonのあとになっている．

HPは，PA-RISCからIA-64への移行は非常に簡単だと言っている．なぜなら，IA-64の命令セットのほとんどはPA-RISCのものであり，バイナリレベルの互換性があるという．これに加えて，データの互換性(エンディアンが同じということ？)もあることが特筆すべきこととして挙げられている．

● Alpha

AlphaはDEC(Digital Equipment Corp.)のAlpha AXPというアーキテクチャに準拠したRISCである．VAXのユーザーをよりハイエンドのEWSへと導くため，1989年の中頃からプロジェクトが始まった．DECにとっては最初のRISCアーキテクチャではなかったが，商業的に初めて成功したのがAlphaである．これは当初から64ビットアーキテクチャを提供し，64ビットのロード/ストア命令を基本として命令セットが構築されている．命令長は32ビット固定で140種の命令がある．最初のAlpha21064は200MHzという，当時としては，信じられない高速動作を実現し，最高速のMPUとしてギネスブックに掲載された．

Alphaは21064(200MHz)に始まり，バイトとワードのデータ型をサポートして動作周波数を向上(最大600MHz)させた21164，新たにモーションビデオ命令(MVI)を追加してさらに動作周波数を向上(700MHz以上)させた21264がこれまでに開発された．この間，「ほとんど誰も欲しがらない最先端技術」の典型と揶揄されながらも，25年のライフサイクルを想定されていたらしい．1999年には量産チップとして初めて1GHz動作を達成した．そして，今後も21364，21464と続いていく予定だった．当初予定では，21364が2000年，21464が2001年に登場する予定だったが，開発は順調に遅れてしまったようである．

なお，Alpha AXPアーキテクチャはCPUコアによって区別される．それはEV(Electro Vlassic)の名称で呼ばれる．その数値はDEC(後期にはIBM)の半導体製造プロセスを表している．

余談ではあるが，AMDはAthlonのバス仕様としてAlpha EV6の仕様を採用してIntelと差別化を図ったのは有名である．

1998年，CompaqはDECを買収してAlphaアーキテクチャを手に入れた．Compaqはx86チップの大口顧客として有名だが，AlphaチップはEWS用である．それまでMIPS社のMPUを使用していたTandem向けEWSを21264で置き換える予定だったようだ．

そして，2001年6月25日，CompaqはAlpha事業をIntelに譲り渡すと発表した．Intelは，Alphaの知的資産を獲得するとともに，Itaniumに対する直接的な脅威を消し去ることに成功した．Compaqは「AlphaのエッセンスはItaniumの中で生き続ける」としているが，Alphaアーキテクチャの事実上の消滅である．

まとめ

ほかにもメジャーどころとしてはSHやV800シリーズなどが，さらにもう少しマイナー路線(？)まで広げると，日本の半導体メーカー各社が何がしかのRISC系マイコンをもっているが，誌面も尽きたので残念だがここまでとする．

Intelに端を発するマイクロプロセッサの歴史は，ほかのアーキテクチャとの攻防があったが，結局はIntelの一人勝ちの状況で進化が進んでいる．プログラムがC言語で開発されるのが主流になった現在では，命令セットアーキテクチャへの関心は薄くなっている．

今後はプロセス技術の進化とともにマイクロプロセッサも進化していく．その先駆者の一つは間違いなくIntelであるが，それと双璧をなすIBMにも，RISCの火を消さないで頑張ってもらいところだ．

なかもり・あきら　フリーライタ

Column

ゲームマシンに採用されたMPUたち

ゲームだから性能が低くても安いMPUでいい……というのはすでに昔の話．現在ではPlayStation2しかり，Xboxしかり，GAMECUBEしかり，少し前のPCを凌駕する性能のMPUが使われている．**表A**にその一部を上げてみる．

〔表A〕ゲームマシンのMPU

ファミリーコンピュータ	6502カスタム
セガ　マークIII	Z80
MSX	Z80
PC-Engine	6502カスタム
3DO　REAL	ARM60
AMIGA500	68000
LYNX	6502カスタム
スーパーファミコン	65816
メガドライブ	68000
PC-FX	V810
バーチャルボーイ	V810
Nintendo64	R4300
セガサターン	SH2 × 2
ネオジオ	68000
ピピン・アットマーク	PowerPC603
PlayStation	R3000カスタム
GAMECUBE	Gekko(PowerPC750カスタム)
Dreamcast	SH-4
PlayStation2	EmotionEngine (MIPS系フルスクラッチ)
Xbox	Pentium III
GameBoy	Z80
ネオジオポケット	TLCS900/H (東芝オリジナル16ビット)
GameGear	Z80コンパチ
ポケットステーション	ARM7T
WonderSwan	V30MX(80186コンパチ)
WonderSwanColor	V30MZ(80186コンパチ)
GameBoyAdvance	ARM7カスタム
PlayStation Portable(PSP)	4K(JadeまたはEmerald)× 2 (MIPS系IPコア)

注：複数のMPUを搭載したものはメインMPUを表記

キャッシュ構造の違いから，680x0/i486/R4000のキャッシュの動作まで

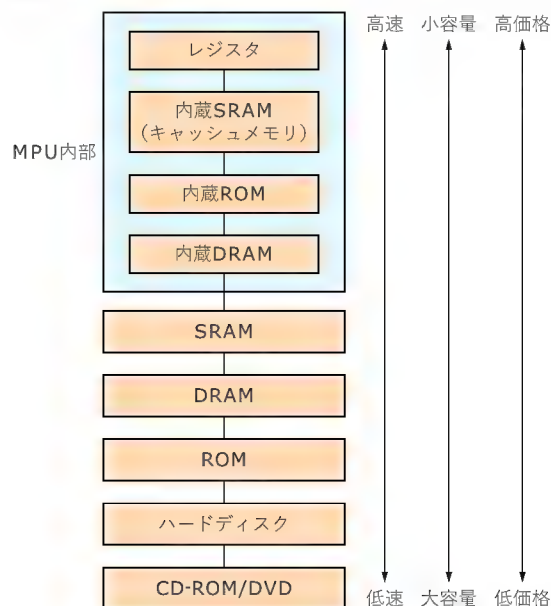
キャッシュのメカニズム

中森　章

一口にキャッシュといっても，フルアソシアティブ/ダイレクトマップ/2ウェイセットアソシアティブなどのライン選択方式，ライトスルー/ライトバックの書き込み制御方式，LRU/FIFO/ランダム方式といったリプレースメント方式など，キャッシュの構造や動作でさまざまな違いがある．ここでは，それぞれのキャッシュ方式の違いを詳しく解説する．（編集部）

はじめに

その昔，フォン・ノイマンがプログラム内蔵方式，つまりプログラムもデータと同じようにメモリ中に格納する方式を提唱して以来，その方式は現在のコンピュータアーキテクチャの基本理念となっている（フォン・ノイマンがプログラム内蔵方式の提唱者というのは正確には誤りだが，ここでは通例にしたがっておく）．Pentiumにしろ PowerPCにしろ，現在でもこの方式から脱却してはいない．当然のことながら，ほとんどすべてのMPUは，プログラムを実行するときにはメモリへアクセスしなければならない．そして，そのメモリへのアクセス時間がプログラムの実行性能にも影響を与えてしまう．これが「フォン・ノイマン・ボトルネック」と呼ばれる現象である．MPUの性能向上のためのキーポイントの一つはフォン・ノイマン・ボトルネックの削減にあるといっても過言ではない．

〔図1〕メモリの階層

● キャッシュメモリとは？

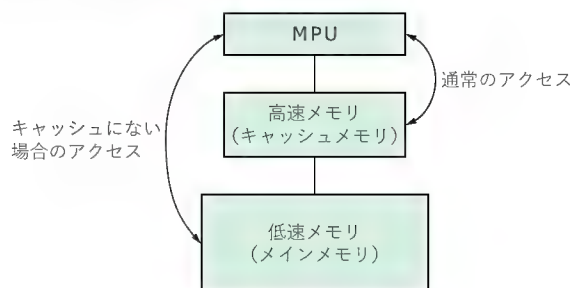
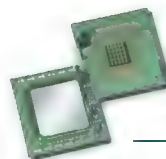
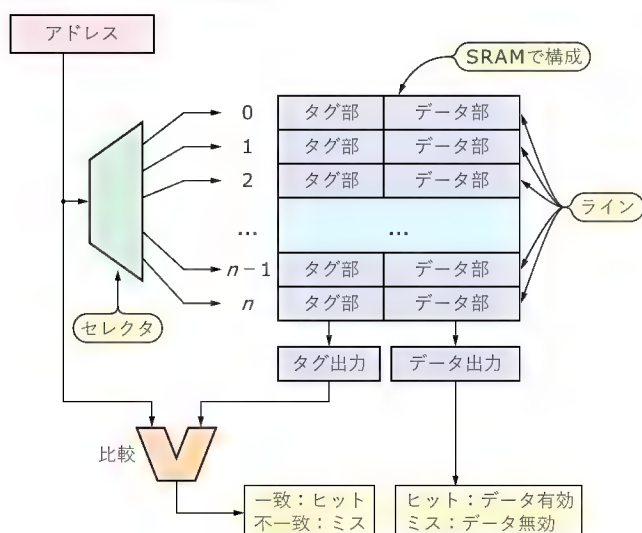
フォン・ノイマン・ボトルネックを削減するための手っ取り早い方法は，高速な（アクセス時間の短い）メモリを使用することである．世の中にはいろいろな種類のメモリ（記憶装置）があり，アクセス時間に応じて**図1**のようなメモリ階層を形成している．高速なメモリは高価であるため，大容量で使用することは難しい．そこで，**キャッシュメモリ**という構造が用いられる．

キャッシュ（cache）とは「隠し場所，貯蔵所」という意味で，キャッシュメモリとは原則としてプログラムで意識する必要のない高速な隠しメモリのことである．具体的には**図2**のように，小容量で高速なキャッシュメモリと，大容量で低速なメモリを階層構造に組み合わせる．

動作としては，低速メモリ（大容量）の内容の一部をキャッシュメモリ（小容量）にコピーしておき，MPUは，通常はキャッシュメモリのみをアクセスする．アクセスすべき内容がキャッシュメモリにない場合は，低速メモリの内容をキャッシュメモリへコピーし直し，そこをアクセスする．このときは低速なメモリからのコピーが発生するので多少時間がかかるが，2度目以降はキャッシュだけにアクセスするので高速となる．たとえばプログラムがループ処理をする場合や，同じ変数を何度も読み書きするような場合，キャッシュへコピーされた命令やデータにアクセスすることになるので，プログラムが高速に実行されるというわけだ．これは，プログラムのメモリアクセスには局所性があるという経験則が基本原理となっている．

〔図2〕**キャッシュメモリの構造**（概念図）

〔図3〕キャッシュの内部構成

〔図4〕フルアソシアティブ方式

キャッシュメモリは単に「キャッシュ」と呼ばれることが多い．本稿でも，以下ではキャッシュと表記する．また，低速メモリからキャッシュへコピーのしなおしはリフィル，またはリプレースと呼ばれる．

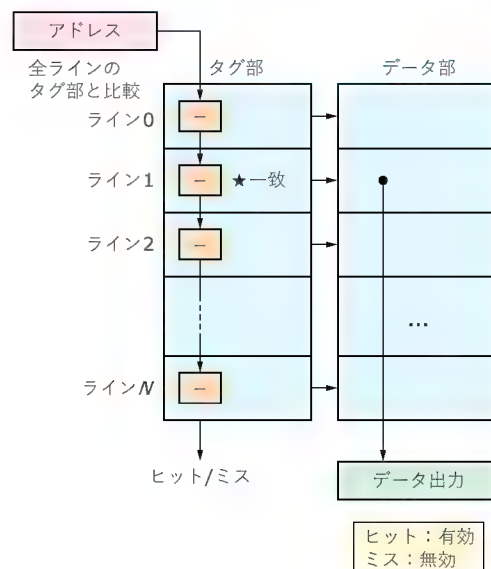
● 昔は外付けSRAMで，現在はMPU内蔵で

現在ではMPUにキャッシュが内蔵されることは珍しくない．しかし，LSIの集積密度がそれほど高くなかった10年くらい昔では，SRAMを使用してMPUの外部にキャッシュを構成していた．とはいえ，SRAM自体が非常に高価だったため，本当に性能の必要な大型計算機などでしかキャッシュは採用されていなかった．ところが，現在主流のRISCではキャッシュの存在を前提とし，メモリへのアクセスは，とりあえずキャッシュヒットするものと仮定してアーキテクチャが決定されている．LSI製造技術の進歩には目を見張るものがある．

なお，本稿ではMPUに内蔵されているキャッシュ，とくに1次キャッシュを念頭において解説しているが，解説そのものはキャッシュについての一般論である．

1 キャッシュの内部構成

● キャッシュの構成

キャッシュは，高速，(比較的)小容量である点を除けば通常のメモリと変わりはない．アドレスを与えると対応するデータが出力される．ただし，低速なメモリ(メインメモリ)の一部をコピーしたものなので，対応するアドレスのデータが格納されていないことがある．これを**キャッシュミス**(あるいはミスヒット)という．このキャッシュミスを検出するため，特殊な構造を採用している．具体的には，タグ部とデータ部と呼ばれるメモリの組(これをラインまたはエントリと呼ぶ)の集合がキャッシュである(**図3**)．各アドレスに対して特定のラインが選択され，そのラインのタグ部の内容が与えられたアドレスに一致すればヒットであり，そのラインのデータ部の内容が与えられたアドレスの内容である(有効)ことがわかる．

逆に，タグ部の内容が与えられたアドレスに一致しなければミスであり，データ部の内容は与えられたアドレスのものではない(無効)．現実にはタグ部の中には，ラインの内容が有効なものであるか否かを表す「バリッドビット」も含まれている．バリッドビットが無効を示していれば，アドレスとタグが一致してもミスとみなされる．

また，データ部の容量はまちまちである．昔は，1ワード(4バイト)の場合が多かったが，現在では4ワード(16バイト)や8ワード(32バイト)が主流である．一般に1ラインのデータ部の容量(バイト数)が大きくなるほど，タグ部に必要なビット数を少なくできる．ただし，データ部の容量を大きくしすぎると，アクセスするアドレス範囲がランダムな場合にキャッシュのヒット率が低下し，性能が低下する．このため，データ部の容量の決定は，予想されるヒット率や利用できる回路規模(この場合は面積)を考慮して決定しなければならない．

● ラインの選択方式(連想方式)

キャッシュではアドレスが与えられると，ある一つのラインが選択される．この方式には，大きく分けて次の3種類がある．

(1) フルアソシアティブ方式
(2) ダイレクトマップ方式
(3) nウェイセットアソシアティブ方式($n \geqq 2$)

● フルアソシアティブ方式

この方式の概念図を**図4**に示す．フルアソシアティブ方式においては，与えられたアドレスはすべてのタグ部の内容と比較される．アドレスとタグが一致するラインが存在すればヒット，存在しなければミスである．**図4**の例ではライン1がヒットしているので，ライン1のデータ部の内容が有効なデータとして出力される．

この方式は直感的にわかりやすく，ラインをもっとも有効利用できる（したがって，同じライン数ではもっともヒット率が高い）方式であるが，全ラインのタグ部との比較のための論理回路が巨大になるため，また，後述する，キャッシュミス時にリフィルするラインを決定するためのLRU（Least Recently Used）処理が複雑になるので，あまり採用されない．

もっとも，LRU処理をあきらめて，FIFO（First In First Out）制御やランダムな選択でリフィルするラインを決定することも考えられる．その場合，ネックとなるのはタグ部の比較論理の回路規模だけである．ライン数が少数（64程度）であれば，連想メモリなどを用いて比較回路を構成することは難しくない．そのため，この方式は，MMUのTLB（Translation Look-aside Buffer）において，仮想アドレスから対応する物理アドレスを選択する（アドレス変換）場合で採用されることが多い．

● ダイレクトマップ方式

この方式の概念図を**図5**に示す．この方式では，与えられたアドレスをデコードして特定の一つのラインに対応させる．デコード，といっても大袈裟なものではなく，単にアドレスの1部分のビット列でラインを選択することが多い．キャッシュの構成が256個のラインからなり，1ラインのデータ部が4ワード（16バイト）だとすれば，現在のMPUではバイトごとにアドレスが割り振られているので，アドレスのビット4からビット11の8ビットで参照するラインの番号を決定すればよい（8ビットなので256種類の値を指定できる）．

もっとも，アドレス内の連続する8ビットで指定した場合，アクセスするアドレス範囲が大きい場合はヒット率が低下する恐れもあるので，アドレスの上位数ビットを考慮したり，アドレスの二つの部分のビット列の排他的論理和を計算して参照するラインを決定する場合もある．

この方式は，キャッシュリフィル時のラインが一意に決定されるのでLRU制御を行う必要がなく，回路構成も単純なため（したがって高速に動作するし消費電力も少ない），1世代前のMPUの内蔵キャッシュに多用されていた．

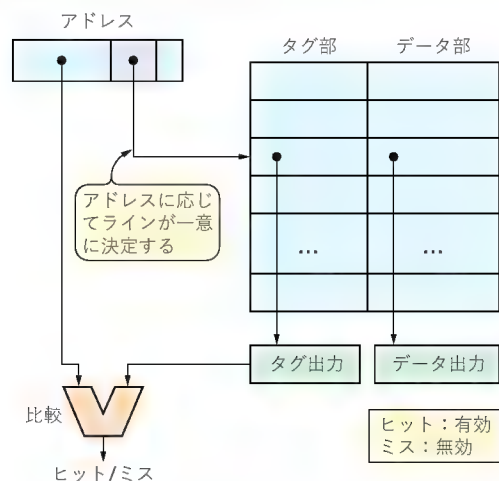
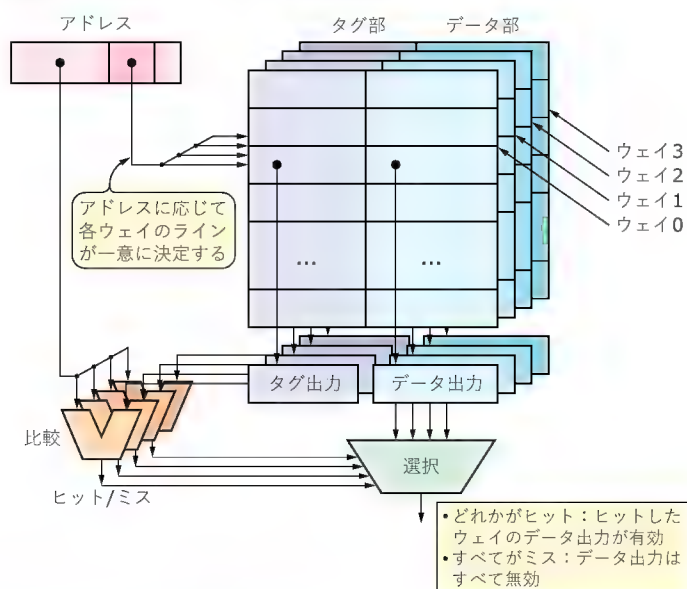
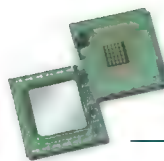
● nウェイセットアソシアティブ方式

この方式の概念図を**図6**に示す（$n = 4$の場合）．見てわかるようにnウェイセットアソシアティブ方式は，ダイレクトマップ方式の構成をn個並列に並べたものであり，それぞれが「ウェイ」と呼ばれる．n個のタグの比較器をもち，アドレスをデコードして決定される各ウェイに属するラインのタグ部出力を同時に比較する．一つでも一致するラインが存在すればヒットである．

この方式は構造が比較的単純で，ダイレクトマップ方式と比べてキャッシュのヒット率を上げることができる（最悪でもダイレクトマップと同じ）ため，もっとも多く採用されている．最新のMPUでは$n = 2$または4で構成されることが多いようだ．nの値を大きくすればするほどキャッシュのヒット率は向上するが，nが十分大きい場合は，nと$n + 1$でのヒット率に大差はない．経験的には，$n = 4$が回路規模とヒット率を考慮した場合の最適解であるとされている．

なお，各ウェイに含まれるライン数が1で，nがラインの総数に等しい場合がフルアソシアティブである．nウェイセットアソシアティブ方式は，ダイレクトマップ方式とフルアソシアティブ方式の折衷案ということもできる．

ところで，インテルのStrongARM（XScale）は32ウェイセットアソシアティブと，驚異的なウェイ数を実現している．これは，ほとんどフルアソシアティブ並みといえる．ARMの文献を読むと，この32ウェイ構造は連想メモリによって実現しているそうである．そうなると，フルアソシアティブとどう違うのかという疑問が沸く．その実装方式は明らかにされていないが，ど

〔図5〕ダイレクトマップ方式

〔図6〕4ウェイセットアソシアティブ方式

うやらフルアソシアティブキャッシュを32分割して，1ウェイあたり64エントリ(キャッシュサイズ16Kバイトの場合)で制御しているようである(64エントリのフルアソシアティブキャッシュが32個ある)．連想メモリがタグの比較も行うので，1ウェイからは1ビットのヒット/ミス信号が出力されるのみである．これは32個のタグを同時に読み出すよりも効率がよさそうである．もっともこれは，仮想アドレスキャッシュ(詳細は後述)だからできる芸当であろう．

● 各方式でのキャッシュの効率

ただし，キャッシュのライン数(＝サイズ)が多いことがキャッシュ効率と直接には結びつかないことにも注意したい．同容量のキャッシュサイズの場合，連続的にキャッシュできるエリア，ないしはウェイごとのキャッシュ容量は，

キャッシュ容量/n

で表される．

ここで，たとえば容量が0x800バイトのnウェイセットアソシアティブ構成のキャッシュを考える．$n = 8$の場合，各ウェイの容量は0x100バイトである．1ラインの容量を16バイトとすると，アドレスのビット7～4(4ビット＝ライン数は16)が各ウェイのラインへのインデックスとなる．そして次のような3種類のアクセスパターンで，キャッシュの効率を見てみよう．

▶アクセスタイプaの場合

さて，プログラムがアクセスするアドレスが，

0x010，0x210，0x410，0x610，
0x810，0xA10，0xC10，0xE10　…(アクセスタイプa)

というパターンで考えてみよう．これは，どれもラインへのインデックスは0x01であり，8ウェイあればすべてのアドレスをキャッシュできる．それでは，$n = 4$の場合はどうだろう．各ウェイの容量は0x200バイトであり，アドレスのビット8～4(5ビット＝ライン数は32)がラインへのインデックスとなる．上の八つのアドレスに対して，この場合もインデックスはすべて0x01となる．したがって，4ウェイでは八つのうちの四つしかキャッシュすることができない．効率は半分に低下する〔**図7(a)**〕．

▶アクセスタイプbの場合

次にプログラムがアクセスするアドレスが，

0x010，0x110，0x210，0x310，
0x410，0x510，0x610，0x710　…(アクセスタイプb)

であるとどうなるだろう．8ウェイの場合は，すべてのインデックスが0x01なので，先の例と同じく，すべてをキャッシュできる．一方，4ウェイの場合は，

0x010，0x210，0x410，0x610

のアドレスに対するインデックスは0x01だが，

0x110，0x310，0x510，0x710

のアドレスに対するインデックスは0x11である．インデックスが0x01と0x11のアドレスが4組あることになるので，4ウェイでもすべてのアドレスをキャッシュできる．この場合のキャッシュ効率は同じである〔**図7(b)**〕．

▶アクセスタイプcの場合

さらに，アクセスするアドレスが次のように偏っている場合を考える．

0x010，0x110，0x210，0x910，
0xA10，0xB10，0x1010，0x1110　…(アクセスタイプc)

この場合は，大まかに2か所にデータが分布している．上と同様に考えると，8ウェイでも4ウェイでも効率は変わらない．しかし，2ウェイだと少しだけ，ダイレクトマップとなると大幅に効率が落ちてしまう〔**図7(c)**〕．

以上の例でわかることは，ライン数よりもウェイ数を増やしたほうが効率的ということである．まあ，そのほうがフルアソシアティブ方式に近くなるので，当然といえば当然である．しかし，アドレスのばらつきがアクセスタイプbの組のような条件ならば，無理して複雑な8ウェイ構成にする必要はない．4ウェイで十分である．また，アクセスタイプcの組のような条件では，キャッシュ構成の複雑さとヒット率のトレードオフを考えると，2ウェイが最適といえる(2か所に分布する傾向があるため)．

● キャッシュサイズの決定

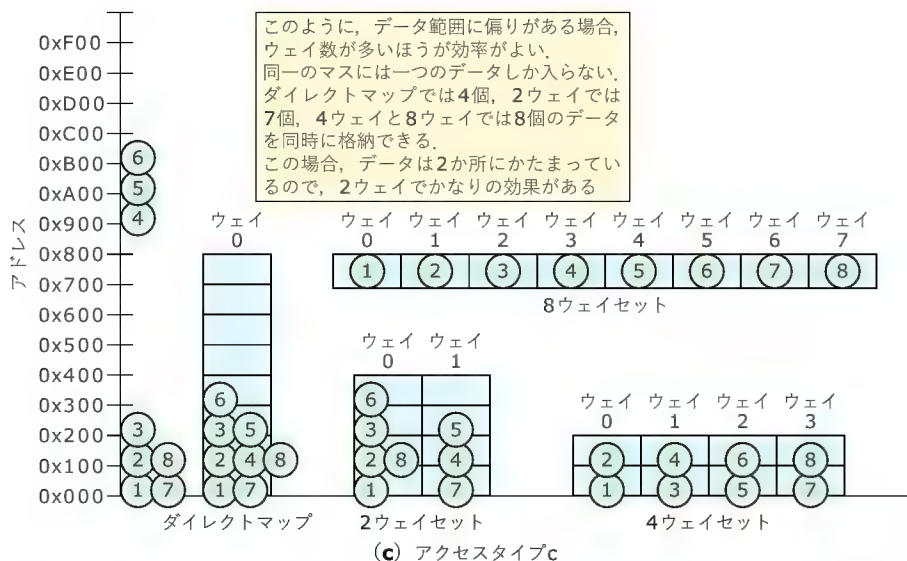
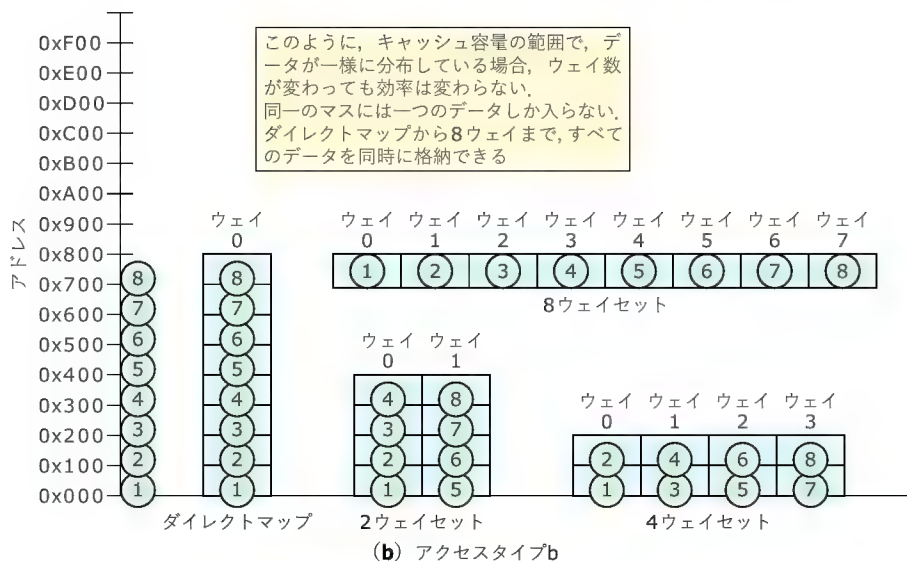
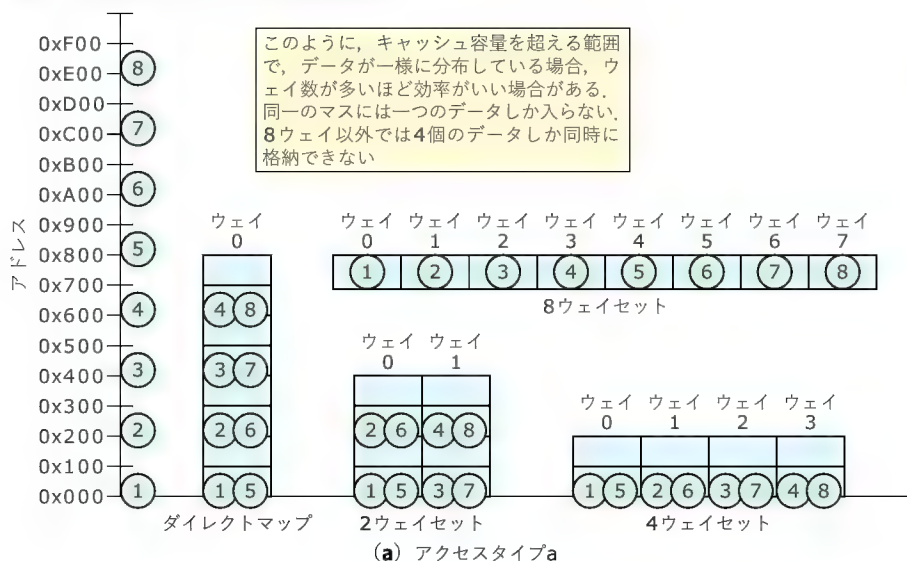
実際のキャッシュ設計において，キャッシュサイズが限定される場合，さまざまなシミュレーションを行ってもっとも効率のよいと考えられるウェイ数に決定される．マルチスレッドで動作するプログラムをキャッシュする場合は，アドレスの下位ビットが一致する確率が高いので，ウェイ数を重視したほうが効率が上がる．Java処理系など，インタプリタやカーネルなどのある程度広がりをもった局所的な部分にアクセスが集中しがちな場合は，ウェイごとの容量が大きいほうが効率が上がる傾向にある．キャッシュ構成の決定には，使用されるであろうOSやプログラムの種類などをよく吟味しなければならない．

以上の性質を直感的にいえば，次のようになる．アクセスするアドレス範囲が真にランダムであれば，キャッシュのヒット率はキャッシュサイズのみで決定する．キャッシュの構成には無関係である．しかし，現実にはアクセスする範囲に偏りがあるので，ウェイに分けたほうがヒット率が上がる．たとえば，通常のアプリケーションプログラムでは，命令はアクセスがユーザー領域とOS領域の2か所に偏る傾向があり，2ウェイセットアソシアティブキャッシュが有効である．あるいは，データはプログラム固有のデータ領域とスタックの2か所をアクセスするので，この場合も2ウェイセットアソシアティブキャッシュが有効である．しかし，現実にはプログラムの動きはもう少し複雑なものと考えられ，経験的には4ウェイセットアソシアティブがもっとも効率的とされている．そうであっても，構成の簡単さ，消費電力の考慮から，2ウェイセットアソシアティブ構成が採られる場合も多い．あるいは，キャッシュサイズがある程度小さい場合は，アクセス範囲が十分ランダムとみなせるため，ダイレクトマップ構成も採用される．

〔図7〕各方式でのキャッシュ効率の比較

（a）アクセスタイプa

（b）アクセスタイプb

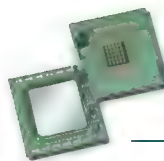
（c）アクセスタイプc

2 キャッシュへのアクセス方式

キャッシュとは，アドレスを与えて（ヒットすれば）それに対応するメインメモリの（コピーしている）データを得るものである．この場合，与えるアドレスが仮想アドレスであるか物理アドレスであるかによって，特徴に若干の違いがある．

● 物理インデックス，物理タグ

この方式は，一般に「物理アドレスキャッシュ」と呼ばれる．物理アドレスからキャッシュのラインを決定し，出力されるタグ部には物理アドレスが格納されているものとして比較する．キャッシュをMPUの外部に取り付けるしかなかった昔では，MPUの外部バスから出力されるアドレス（もちろん物理アドレス）でキャッシュにアクセスするしかないので，当然物理アドレスキャッシュである．次に述べる仮想アドレスキャッシュと違い，タスク切り替えごとにキャッシュを無効化する必要がないので，制御が簡単である．しかし，仮想アドレスから物理アドレスへのアドレス変換が終了しないとキャッシュにアクセスすることができないので，キャッシュのアクセス時間に余裕がなくなり，高速動作させることが難しいのが欠点である．

● 仮想インデックス，仮想タグ

この方式は，一般に「仮想アドレスキャッシュ」と呼ばれる．仮想アドレスからキャッシュのラインを決定し，出力されるタグ部には仮想アドレスが格納されているものとして比較する．この方式ではアドレス変換と同時にキャッシュにアクセスできるため，また，キャッシュ自身にタグ比較の論理を取り込むこともできるため，キャッシュアクセスに余裕ができ，高速で動作させることが可能である．しかし，欠点もある．メインメモリへの最終的なアクセスは物理アドレスで行われるので，メインメモリのデータは物理アドレスで一意に区別できる．つまり，物理アドレスが同じなら同じ場所，物理アドレスが異なれば異なる場所

を指す．しかし，仮想記憶で動作している場合，仮想アドレスが同じでも，同じ物理アドレスを指し示しているとは限らない（ほとんどの場合，異なる物理アドレス）．ということは，単純に考えると，仮想アドレスだけでタグ比較を行っていると意図した物理アドレスと異なる場所からデータを取ってしまうことがある．これをエイリアシングまたはシノニムの問題という．

通常，仮想アドレスと物理アドレスの対応はタスクごとに決まっているので，タスクが切り替わるとキャッシュのタグ部に格納されている仮想アドレスは無意味なものになる．したがって，仮想アドレスキャッシュを採用する場合は，タスク切り替えごとにキャッシュの内容を無効化する必要がある．これは制御回路の増大を招く．これを防ぐ方法としてタグ部の中にタスクIDを一緒に格納しておき，タグの比較時にタスクIDも比較することが考えられる．しかし，この場合はタグ部のビット数が増大する．また，ごく稀ではあるが，異なる仮想アドレスに同一の物理アドレスを対応させる場合もある．仮想アドレスキャッシュはこの場合に対応できない．

インテルのStrongARMは仮想アドレスキャッシュを採用している．最初のSA-110はタスクIDをサポートしていなかったが，これでは実用性に乏しいのか，Windows CEに採用されたSA-1100やSA-1110ではタスクIDをサポートするようになった．

● 仮想インデックス，物理タグ

この方式にはとくに決まった呼称はない（と思う）．仮想アドレスからキャッシュのラインを決定し，出力されるタグ部には物理アドレスが格納されているものとして比較する．これは，物理アドレスキャッシュと仮想アドレスキャッシュの折衷案である．アドレス変換と同時に仮想アドレスでキャッシュにアクセスし，アドレス変換が終了する頃に，キャッシュから出力される物理アドレスとアドレス変換した物理アドレスを比較する．キャッシュのアクセス時間に余裕ができ，タスク切り替え時の無効化も必要ない．この方式はMotorolaのMC68040以降やMIPS系のMPUで採用されている．

3 リプレースメント方式

キャッシュはヒットすることが前提とはいえ，現実には頻繁にミスが発生する．この場合，キャッシュ内にメインメモリの新しいコピーをもってくる必要がある．このとき，どのラインに新しいデータを書き込むのかを決定する方法が**リプレースメント方式**である．書き込むラインが決定すれば，そこに新しいデータをリフィル（リプレース）する．ダイレクトマップ方式の場合は何の考慮も必要ない．アドレスに対して対象ラインは一つしかないので，そこをリフィルする．nウェイセットアソシアティブの場合は，与えられたアドレスに対して対象ラインはn個あるので，それから一つを選択しなければならない．フルアソシアティブの場合は，すべてのラインがリフィルの対象である．

● LRU（Least Recently Used）方式

この方式は，プログラムの（時間的な）局所性という経験則に依っている．すなわち，いちばん昔にアクセスされたラインはこれからアクセスされる確率が低いのでそこを更新する，というもっとも妥当な方式である．この方法では，nウェイセットアソシアティブ方式の場合は，各ウェイの同一インデックスにあるn個のラインに対するアクセス頻度の履歴を記憶しておく．そのために，$n=2$の場合は1ビット，$n=4$の場合は6ビット，$n=8$の場合は28ビットのメモリが必要である．フルアソシアティブの場合は全ラインのアクセス頻度の履歴を記憶しなければならないので，ほとんど非現実的なビット数のメモリが必要である．このため，LRU方式は，主としてnウェイセットアソシアティブ方式で用いられる．

この方式の欠点としては，ラインへのアクセス（ヒット）ごとにLRUメモリを更新しなければならないので，タイミング的に厳しいということくらいだろうか．

● FIFO（First In First Out）方式（ラウンドロビン方式）

この方式は，nウェイセットアソシアティブ方式において，0，1，2，…，$n-1$，の順にリフィルするラインを決定するものである．キャッシュラインがすべて無効な状態からリフィルを続けて行くと，ウェイは，0，1，2，…，$n-1$の順にリフィルされていくので，この順に古いデータが格納されているとみなして，その順序で新しいラインを決定する方式である．アクセス頻度が無視されているが，一応，古いラインからリフィルしていくという方針である．ヒットする場合に順序の更新が行われないので，当然LRU方式よりもヒット率は悪くなる．履歴の記憶に必要なメモリのビット数は，カウンタを形成すればいいので，$n=2$の場合は1ビット，$n=4$の場合は2ビット，$n=8$の場合は3ビットで足りる．LRU方式に比べて少ないビット数で済むのが特徴である．フルアソシアティブ方式の場合はラインの番号順にリフィルしていけばよいだろう．

先にも挙げたが，インテルのStrongARM（SA-1100）は，32ウェイセットアソシアティブという（嘘のような？）キャッシュ構成を採っているが，さすがにLRU方式ではなく，このFIFO方式を採用している．FIFO方式は，対象エントリの番号が順次回転していく（最後の次は最初に戻る）ので，ラウンドロビン（回転）方式ともいう．

● ランダム方式

この方式は，ランダム（無作為）にリフィル対象のラインを決定する方式である．どのアドレスも同じような頻度でアクセスされるはずという予測に基づいている．ラインを指定するために必要なメモリのビット数はFIFO方式の場合と同じである．1クロックあるいはキャッシュへの1アクセスごとにそのメモリを更新（たとえば+1）しておいて，リフィルが必要になった場合に，そのメモリの値が（たまたま）示しているラインをリフィルする．ヒット率としてはFIFO方式と大差ないと思われる．論理が単純なためか，この方式はけっこう多くのMPUで採用されているようである．

4 書き込み制御

キャッシュは何もリードするだけではない．書き込みを行う場合もある．キャッシュはメインメモリの内容をコピーしているものだから，常にメインメモリの内容と整合性（コヒーレンシ）が保たれている必要がある．それを実現するために，いくつかの制御方式が考案されている．

● ライトスルー（ストアスルー）方式

これはライトデータに関して，常にメインメモリにも書き込みを行う方式である．誰もが考えつく方式であろう．ライトアドレスがキャッシュにヒットする場合は，ライトデータをメインメモリと同時にキャッシュのデータ部にも書き込む．キャッシュミスの場合はキャッシュは無視してメインメモリのみにデータを書き込む方式が一般的である．

キャッシュミスの場合には，まずリフィルを行い，そのラインとメインメモリの両方にデータを書き込む方式もある．これは，ライトアロケートと呼ばれる．スタックなど，ライトしたアドレスは再びリードする傾向があるので，あらかじめそこのアドレスをキャッシュに入れておこうという発想である．ライトアロケートは，ライトしたアドレスを再びリードする確率が高くないと効果がない．ライトしたアドレスを再びリードする場合も，後で発生するはずのリプレースをライト時に先行して行うだけなので，トータルのリプレース回数には変化がない．この意味で，ライトアロケートが効果的かどうかという点については疑問が残る．

ライトスルー方式を採用する場合，ライトごとにメインメモリへの書き込みバスサイクルが発生するので，連続してライトを行う場合は，前の書き込みバスサイクルが終了するまで次の書き込みバスサイクルを開始できない．このときMPUのパイプライン処理が待ち合わせのために停止してしまう．それを防ぐために，ライトスルー方式を採用するMPUではライトバッファを数段分もっていることが多い．逆に，ライトバッファがないと性能が低下する．

〔図8〕ライトバックキャッシュの状態遷移

● ライトバック（コピーバック）方式

この方式は，メインメモリへのライトアクセスを最小限に抑える方式である．つまり，ライトが発生しても（ヒットする場合は）キャッシュのデータ部のみしか更新しない．当然，メインメモリとの整合性は保たれなくなる．その代わり，そのラインの整合性が保たれていないことを記憶しておく．そして，後でラインごと一括してメインメモリに書き戻す．そのタイミングは，そのラインがキャッシュにミスし，新しいデータをリフィルしなければならないときである．ライトのいくつか（大半？）はキャッシュにヒットするので，メインメモリに対する書き込みバスサイクルの回数を削減することができる．このメインメモリへの一括した書き込み動作を特別に**ライトバック**と呼ぶ．

ライトバック方式のキャッシュはライトアロケートである．キャッシュミスが発生すると，まずリフィルを行って，そのラインのデータ部にライトデータを書き込む．このとき，メインメモリには書き込まない．また，ライトバック方式のキャッシュでは各ラインが現在のキャッシュ状態というものをもっている．メインメモリと整合性が保たれている状態をクリーン（Clean），整合性が保たれていない状態をダーティ（Dirty）という．この状態を示す情報はタグ部に格納されている．**図8**にライトバック方式のキャッシュの状態遷移を示す．

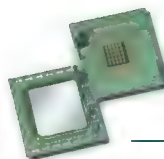
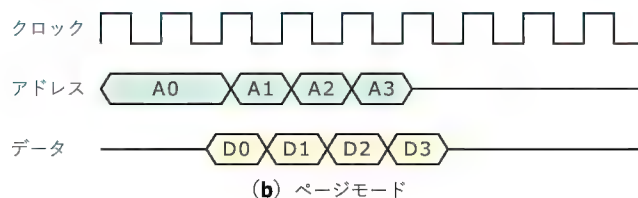
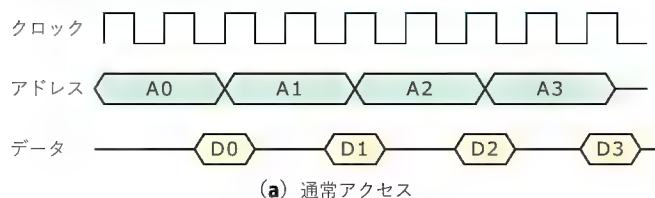
ライトバックはライン単位で行われるのが通常である．つまり，1ラインごとに1ビットのダーティビットをもって管理する．しかし，ラインのすべてがダーティになるのは稀である．たとえば，1ラインが32バイトだとすると，そのうち4バイト程度しかダーティにならないことがある．これを中途半端なダーティという．この場合，1ラインの32バイトすべてをライトバックするのは効率的でない．本当にダーティな4バイトのみをライトバックできれば，ライトのバスサイクルが減少するので，メモリ効率が良い．これを実現するには，1ラインあたりのダーティビットを複数もつことである．たとえば，StrongARMは，8ワード（32バイト）の1ラインに対して，下位4ワード用と上位4ワード用の2ビットのダーティビットをもつ．キャッシュのリフィルは必ず8ワードで行われるが，ライトバックは，ダーティビットの状況に応じて，4ワードまたは8ワードで行う．

5 キャッシュを支える各種機能

● リフィルサイズ

キャッシュミスが発生すると，そのラインはリフィルされる．リフィルは通常ライン単位に行われる．たとえば，ラインのデータ部が4ワード（16バイト）なら，一度に4ワードのデータをメインメモリから読み込む．これは，いったんアクセスしたアドレスの近傍を再びアクセスする確率が高いという，またもやプログラムの局所性に依っている．また，キャッシュのリフィ

〔図9〕バースト転送のイメージ

(a) 通常アクセス

(b) ページモード

ル時に発生するバスサイクルは一般にバースト転送と呼ばれるバスサイクルである．これは，メモリをバスクロック同期で連続的にアクセスする．最近のメモリデバイスはRAMにしろROMにしろページモードというモードをもっている(今流行のSDRAMも似たような動作をする)．このモードにおいては，最初のアクセスのアクセス時間はやや遅い(というか通常の速さである)が，連続するアドレスの2回目以降は，最初の半分程度のアクセス時間でアクセスできてしまう．4ワードのデータを4回に分けてリードするよりも，4ワードのバースト転送を行ったほうがはるかに高速なのである(**図9**)．

MPUによっては複数のラインを同時にリフィルするものもある．これは，アクセスする可能性が高いアドレス範囲をあらかじめキャッシュに入れておくほうがヒット率の向上が見込めるためだが，ページモードとの相性のよさも考慮されているはずである．

現在のMPUでは，1回のリフィル時にリードするデータ量(ラインのワード数，または，その倍数)は8ワードが多いようである．MPUによっては32ワード程度まで設定可能なものもある．プログラムの性質(分岐の発生頻度や同じアドレスをアクセスする確率の大小)を考慮しながら，最適な値をユーザーが設定できる．

● ノンブロッキングキャッシュ

通常，キャッシュミスが発生すると，リフィル動作(バースト転送)が終了するまでパイプラインが止まってしまう．ノンブロッキングキャッシュとは，キャッシュミスが発生してもパイプラインを停止せずに先に進める技術である．キャッシュミスをヒットのように扱うことから「ヒットアンダーミス」ともいう．

具体的な実装は，リフィルデータを格納するためのリードバッファを何組か用意しておき，キャッシュミスが発生するとリードバッファとリード(またはストア)を発生する命令を関連づける．リフィルはリードバッファに対して行い，キャッシュは暇を見て更新する．その間パイプラインを止めるようなことはしない．

さらに，リフィル要求と同時にバッファのID(番号)を同時に出力し，外部からはデータにそのIDをつけて返してもらう方式も考えられる．リフィルデータはIDで区別できるので，キャッシュミスを発生した順序でデータを返す必要はない(アウトオブオーダ)．もちろん，キャッシュミスを起こした順番にデータを返す(インオーダ)場合は，データを区別するIDは不要である．2次キャッシュをもつ場合や，マルチプロセッサ構成になると，アクセスごとにデータを用意できる時間が異なるので，アウトオブオーダなデータ応答は実効性能を上げる意味がある．**図10**にノンブロッキングキャッシュの概念図を示す．

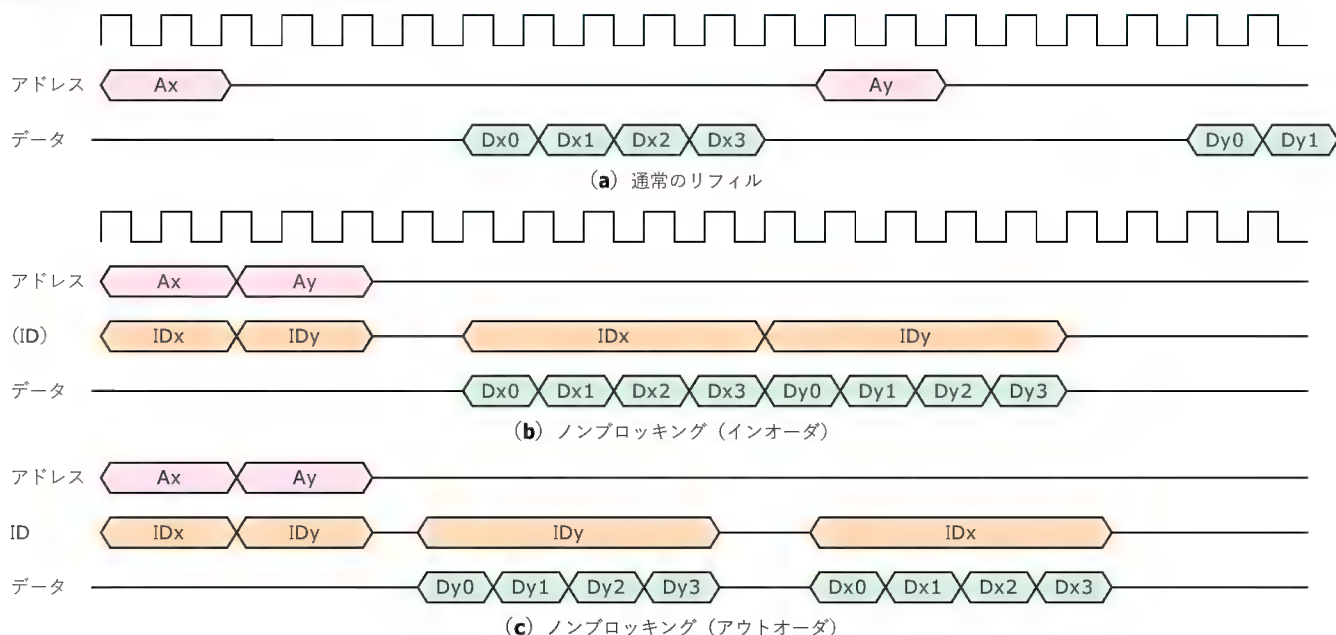
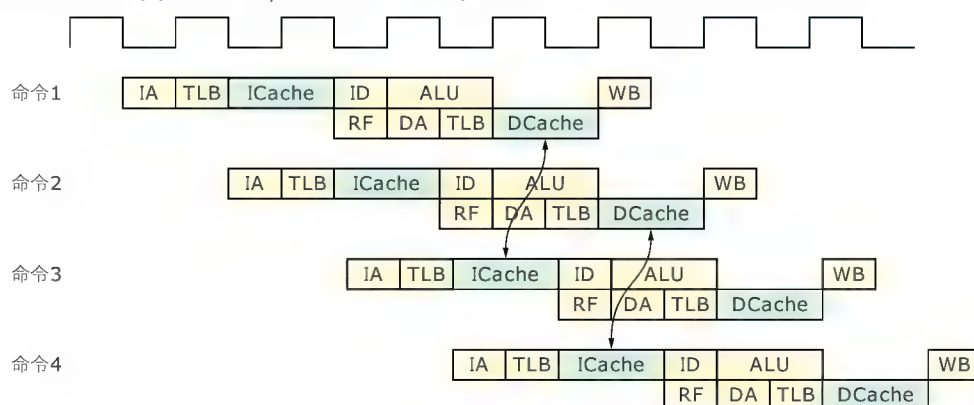
ノンブロッキングキャッシュは，リードしたデータをすぐに参照しない場合に効果がある．このためにはコンパイラの命令スケジューリングによる最適化が必要になる．

● 命令キャッシュとデータキャッシュ

図11にR3000のパイプライン動作を示す．この図で，「ICache」が命令キャッシュへのアクセスを示し，「DCache」がデータキャッシュへのアクセスを示している．図を見ると命令1のデータキャッシュへのアクセスと命令3と命令4の命令キャッシュへのアクセスのタイミングが重なっている．命令キャッシュへのアクセスは毎回発生するが，データキャッシュへのアクセスはロード/ストア命令のみで発生するので，アクセスが重なることは多くないが，まったくないとはいえない．この場合，同じキャッシュからデータを参照することは(どちらかのアクセスを待ち合わせてパイプラインを一時停止しなければ)不可能である．R3000はパイプラインをできるだけ停止させないことを信条としているので，命令キャッシュとデータキャッシュを分けて独立なアクセスを可能にしている．このように，命令キャッシュとデータキャッシュをアクセスする経路を別々に設けるアーキテクチャを(**修正**)**ハーバードアーキテクチャ**という．ハーバード大学で初めて提唱されたのでこの名称があるのだろう．CISCではモトローラのMC68020辺りで初めて採用されたように思う．

ハーバードアーキテクチャの欠点(？)は，命令キャッシュとデータキャッシュが別なので命令書き換えに対応できないことである．また，同じアドレスの内容を命令とデータキャッシュでそれぞれ独立に記憶する場合があるので，メモリのむだといえばむだである．

逆に，命令とデータを同じキャッシュをもつのが**ユニファイドキャッシュ**である．インテルのi486あたりまでがこの方式を採用している．命令書き換えに対応できる(パイプライン動作をしているので，書き換えを行ってからキャッシュに反映されて命令フェッチできるまでに数命令分の遅れがあるはずであるが)し，メモリもむだにならない．i486はハードウェアアーキテクチャこそRISCであるが，命令セットアーキテクチャは「バリバリの」CISCなので，メモリアクセスが非常に多い．命令キャッシュとデータキャッシュの同時アクセスによるパイプライン停止が頻繁に発生していると思われるのだが，どのように対

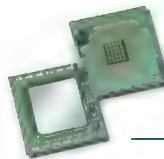
〔図10〕ノンブロッキングキャッシュの概念

〔図11〕R3000のパイプラインと命令キャッシュ/データキャッシュを参照するタイミング

応しているのだろう（詳細情報は公開されていないようだ）．

インテルも，Pentium以降は命令キャッシュとデータキャッシュを分離した．命令キャッシュとデータキャッシュのアクセスの競合をなくすためという．ただし，これまで動いていたプログラムが動かなくなっては互換性に問題を起こすので，命令書き換えは依然としてサポートしているようである．

なお，ハーバードアーキテクチャに対応して，命令とデータの経路が共通な方式を**プリンストンアーキテクチャ**ということもある（あまり一般的ではないが）．これは，初期のコンピュータを提唱したフォン・ノイマン教授がプリンストン大学に属していたことに由来する．

● 1次キャッシュと2次キャッシュ

図1で示したメモリ階層がMPUの内蔵キャッシュにも当てはまる．チップに内蔵できるキャッシュ（1次キャッシュ）の容量にはチップサイズから来る上限値がある（64K～128Kバイト程度）ので，少し低速で大容量（128K～4Mバイト程度）のSRAMをキャッシュ（2次キャッシュ）として外付けする構成が考えられる．この場合，外付けという性格上2次キャッシュは物理アドレスキャッシュである．MIPSのR4000/R5000/R10000などは，この構成を採用している．また，2次キャッシュがチップに内蔵されるようになった最近では，外付けの3次キャッシュをサポートするMPUも登場してきている．

さて，1次キャッシュと2次キャッシュは階層構造をもっているなら，1次キャッシュの内容は2次キャッシュにも格納されていることになる．1次キャッシュと2次キャッシュの内容を重複させないほうが，よりキャッシュの容量を活かせるのではないだろうか．このようなキャッシュをThunderbirdやDuron（Spitfire）に採用したのがAMDで，同社では従来方式をインクルーシブ（inclusive）キャッシュ，重複させない方式をイクスクルーシブキャッシュと呼んでいる．

さらに1次キャッシュ(とくにダイレクトマップ構成の1次キャッシュ)のヒット率を向上させるしくみとしてビクティムキャッシュがある．これは4～5エントリからなる小規模のフルアソシアティブキャッシュで，1次キャッシュからリプレースで追い出されたキャッシュラインを保持している．

1次キャッシュを参照する際，ビクティムキャッシュも同時に(あるいはビクティムキャッシュを優先的に)参照して，そこにヒットすればビクティムキャッシュからデータを供給する．ビクティムキャッシュのエントリは，基本的にはLRU制御をされ，1次キャッシュから追い出されたキャッシュラインはビクティムキャッシュのもっとも参照されていないエントリに格納される．つまり，ビクティムキャッシュは追い出されたキャッシュラインのうちで最近参照された4～5ラインを保持することになる．これらのラインは，直前にリプレースされた1次キャッシュのラインがもっとも最近参照されたものだが，それ以外では1次キャッシュの他のラインよりも最近参照されたものである場合もある．

● プリフェッチ

プリフェッチとは特定の命令(プリフェッチ命令)を実行することで，パイプラインを止めることなく，キャッシュ(通常はデータキャッシュ)へのリフィルを強制的に行う．同時に，データキャッシュへのアクセスが発生するリード命令やライト命令を実行しない限りパイプラインを止める必要はない．ただし命令キャッシュは，基本的には，絶えずアクセスされているので，パイプラインを止めずに命令キャッシュへのプリフェッチを行うことは事実上不可能である．したがって，命令キャッシュへのプリフェッチ命令は，もし存在しても，意味がない．

さて，どの領域をプリフェッチするかはプログラマ(やコンパイラ)が明示的に指定する必要がある．近い将来アクセスする領域を指定しておけば，データキャッシュアクセスと競合しない限り，バスのアイドル期間を縫ってキャッシュへのリフィルが行われる．プリフェッチは有効に使えばかなり効果がありそうである．

プリフェッチが行われる契機はプリフェッチ命令によることが多い．しかし，最近ではハードウェアで自動的にプリフェッチを行う場合もある．キャッシュの無効な部分をそのままにしておくのはもったいないので，できるだけ有効データを取り込んでおこうという考え方である．ハードウェアプリフェッチを実装すれば，プリフェッチ命令を用いなくても，バスのアイドル時間を縫って自動的にプリフェッチすることが可能である．この機構はとくに命令キャッシュに対して有効である．上述したように，命令キャッシュは絶えずアクセスされるので，プリフェッチの契機となるアイドル時間は発生しにくい．命令キャッシュのプリフェッチを効率的に行うには，リードしながらライト可能な機構をキャッシュに埋め込む必要がある．

ただし，命令キャッシュへのプリフェッチは無条件に連続して行えばいいというものではない．実行する命令列には定期的に分岐命令が出現し，まったく別のアドレスに分岐する可能性もある．分岐命令の次までもどんどんプリフェッチするのは効率が悪い．そこで，命令フェッチ部分にプリデコード機能を設け，分岐命令と思われる命令コードに行き当たるとプリフェッチを停止する方式が採用される．あるいは，分岐予測機能もプリフェッチ機構に含め，分岐命令に行き当たっても，分岐予測をしながら，予測した分岐先からプリフェッチを継続する場合もある．この考えを推し進めていくと，Pentium4が採用している実行トレースキャッシュになる．

● フェッチバイパス

多くのMPUはメモリアクセスがキャッシュにヒットすることを前提に設計されている．ノンブロッキングキャッシュは別であるが，キャッシュミスが発生するとリフィルが完了するまでパイプラインが停止する．命令の連続実行という観点でいうと，一度止まってから最高速で動き，また止まってから最高速で動く……という動作を繰り返しているというイメージであろうか．

そこで，誰もが思いつくのが，止まっている時間がもったいないので，リフィルしているデータをキャッシュに書き込むと同時にMPUにも渡してしまうという方式である．そうするとリフィル中もパイプラインが動作できる．ただし，その間は，命令の実行スピードはバスクロック程度になってしまう．これが**フェッチバイパス**である．

パイプラインクロックとバスクロックに差がありすぎる場合は，バイパス効果はあまり期待できないが，差がほとんどない場合は非常に有効である．バスサイクルは常に起動されているわけではなく，バスサイクルとバスサイクルの間には数クロックのアイドル期間が生じる．リフィルする命令数が少ない場合は，この数クロックの間にそれらの命令を実行できてしまう．つまり，このような場合は，バスサイクルと同時に命令を実行するのも，命令をキャッシュに取り込んでから命令を実行するのも，ほとんど同じ実行効率となる．

MIPSでは，R3000の命令実行においてフェッチバイパス方式を採用しており，「命令ストリーミング」と呼んでいる．

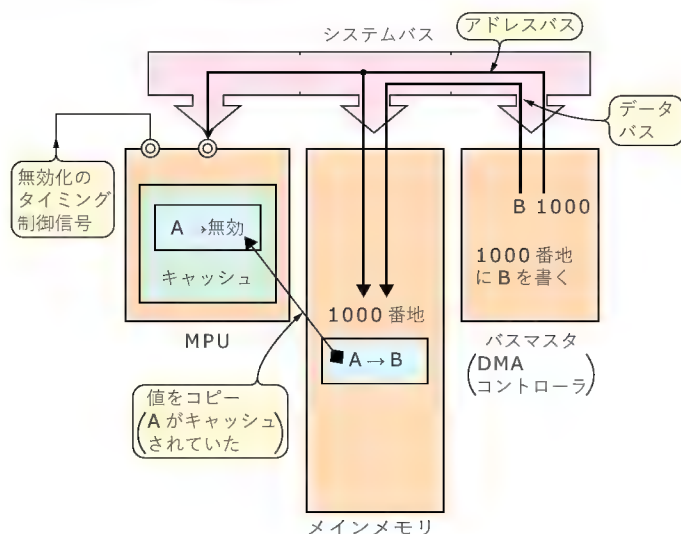
● バススヌープ

DMAなどメインメモリに直接アクセスする処理を行う場合，メインメモリとキャッシュの内容が食い違う現象が発生する．DMAによるデータはI/Oとは異なり，通常はキャッシュしても構わないデータであるが，食い違いをMPUに通知し，メインメモリとキャッシュの整合性を回復する必要がある．このための一手法がバススヌープである．バスモニタともいう．

具体的には，アドレスを指定してそのアドレスにヒットするキャッシュラインを無効化する．この場合，MPUの外部からキャッシュを無効化するアドレス(多くの場合，アドレスバスが使用される)を入力し，専用端子をアサートすることでスヌープが実現される．

図12にバススヌープ機能の概念図を示す．しかし，バススヌープ機能をもたないMPUも多い．DMAコントローラは転送の終了時にTC(Terminal Count)割り込みを発生するので，

MPUはその割り込みを検知して割り込みを発生し，割り込みハンドラ内でDMAされたアドレスに対応するキャッシュラインを専用命令（キャッシュを内蔵するMPUにはたいてい用意されている）で無効化すれば事足りるからである．

このようにDMAの場合は割り込みによってソフトウェアで処理できるが，メインメモリを共有するマルチプロセッサ構成では，他のプロセッサがメインメモリの内容を書き換えたのを検知するのは容易ではない．この場合は，割り込みを使用すると処理が繁雑になるので，バススヌープが活用される（というか，バススヌープ機能がなくてはマルチプロセッサ対応とはいえない）．

〔図12〕バススヌープの概念

〔図13〕ウェイ予測の概念

● ウェイ予測

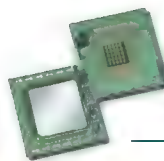
MPUの設計において，キーポイントの一つが消費電力の削減である．MPUの中でもっとも電力を消費する部分は，じつはキャッシュであり，総消費電力の半分程度がキャッシュで消費されているといっても過言ではない．消費電力を削減するために，キャッシュの回路設計においてはメモリセルのブロック分割などの手法が採られることが多い．

そして，ウェイ予測もキャッシュの消費電力を削減するために考案された技術である．対象となるのはnウェイセットアソシアティブ構成のキャッシュである．通常の構成では，あるアドレスが与えられたときに，すべてのウェイのタグ部とデータ部の内容を内部バスに出力し，キャッシュヒットするウェイがあればそこのデータを選択する．

いま，一つのウェイから1回に出力されるデータが1ワード（32ビット）であるとしよう．このとき，4ウェイセットアソシアティブの場合は4ワードのデータが同時に内部バスに出力され，128ビット分の値が変化する．バスを構成する各ビット線を0から1，または1から0に変化させるためにはトランジスタによって目的の値になるようにビット線を駆動しなければならない．このとき電力を消費する．バス上の値が変化しなければ電力はほとんど消費されない．

さて，ウェイ予測とは，内部バスを同時に駆動するのではなく，予測したウェイから順番に駆動していく（キャッシュのヒット/ミスも順番に判定する）方式である．上の例でいえば，1回あたりのバス上の信号変化は32ビット分のみになり，単純計算で，消費電力は1/4になる．ただ，片方のウェイのヒット/ミスを判断してから他方のウェイをアクセスするため，キャッシュアクセスのタイミングは厳しくなるという欠点がある．

図13に，2ウェイセットアソシアティブ構成時にウェイ予測を行う場合のタイミングチャートを示す．図でX，Yはウェイのどちらかを表している．どちらのウェイから先にヒット/ミスの判定を行うか（これが予測）については，LRUビットの値から予測する，前回のキャッシュアクセスと同じウェイを見るなどの方法が考えられるが，決定版という方法はないようである．ウェイ予測が当たればヒット/ミスの判定にロスはないが，予測がはずれればヒット/ミスの判定に1クロック程度のロスが生じる．この場合，たしかに性能は若干低下するが，性能と消費電力のどちらに重点を置くかで，ウェイ予測の採用/不採用が決まるであろう．

事実，最近のMPUではウェイ予測を採用することがけっこうあるようだ．スーパースカラ方式のMPUではデコードした命令を命令キュー（FIFO）に蓄えておき，そこから命令実行ユニットに命令を発行する．命令デコードと命令発行の間には時間差があるので，ウェイ予測ミス時のペナルティは命令キューで緩衝され見かけ上ゼロになる．

ウェイ予測に関しては，基本特許が多く出願されている．最近，ウェイ予測を公表するMPUが多いが，特許の利権関係はどうなっているのだろうと他人事ながら心配してしまう．

ところで，キャッシュアクセスはプログラムの実行においてもっともクリティカル（時間がかかる）部分である．この部分にウェイ予測を導入するとロジックが複雑になり，クリティカルパスを生じやすい．先に予測したタグを見てウェイのヒット/ミスを判断してから初めて別のウェイを参照することは，時間的（RAMのアクセスタイム）に厳しい．したがって，動作周波数を向上したい場合は，ウェイ予測は敬遠される傾向にある．

● 高性能MPUの命令キャッシュアクセス

最近の高性能MPUでは，命令キャッシュへのアクセスとデコード部分を実行部分と切り離して，自律して動作させる．これを**デカップル**（decouple=分離）**方式**と呼ぶ．

つまり，命令実行パイプラインとは無関係に，命令を絶えずメモリから読み込み続けて命令キャッシュに格納している．この際，メモリから出てくるデータをプリデコード（おおまかなデコード，正確である必要はあまりない）して，分岐命令を探し当て，分岐予測機構と共同して，次にアクセスするキャッシュラインを予測する．この機構を**図14**に示す．

デカップル方式では，デコード以降の命令実行パイプラインから見れば，欲しい命令は必ず命令キャッシュにヒットすることを期待している．これは，キャッシュを前提としたRISCでは当然の発想であるが，予測して命令を取り込み続けるフェッチ機構は複雑なので，高性能なMPUでしか採用されない．

デカップル方式で，デコード部分をフェッチ側に見るか，実行側に見るかは微妙なところがある．構造的にはデコード部はフェッチ側に近く，一般にデカップルと言えば，デコード部と実行部以降が命令キュー（リザベーションステーション）の前後で分離されていることを指す．

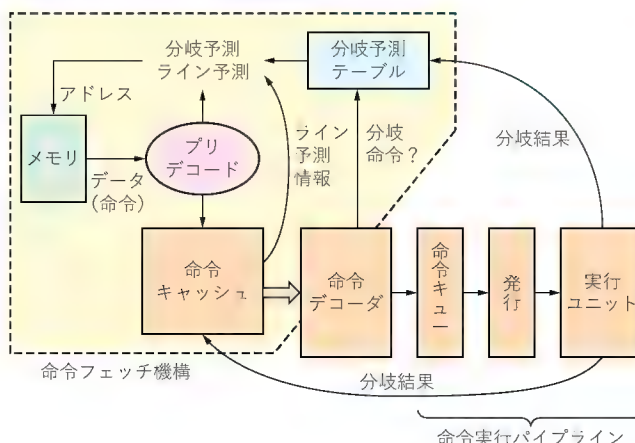
デカップル方式というか命令フェッチ機構の自律化は，今は亡き（?!）Alpha，MIPS R10000シリーズ，PowerPCが採用している．Pentium4に採用された実行トレースキャッシュも，似たような発想である．

● 仮想アドレスキャッシュは最近の流行か

仮想アドレスキャッシュは，動作周波数の高い最近のMPUでは流行になりつつある．キャッシュのアクセスが周波数向上のクリティカルパスとなることは稀ではないので，TLBを参照せずにヒット/ミスを決定できる仮想アドレスキャッシュは，キャッシュアクセスに余裕をもたせることができる．

最近では，日立/STマイクロのSH-5が全面的に仮想アドレスキャッシュを採用した．すでに記述したように，仮想アドレスキャッシュには，マルチタスク下において，同じ仮想アドレスで違う物理アドレスを指し示す場合がある（シノニムの問題）という欠点がある．

一般的にはプロセスID（タスクID）をキャッシュタグに付加することでシノニムの問題を解消する手法が採られるが，SH-5

〔図14〕命令フェッチの自律機構

ではキャッシュミス時にTLBを参照し，その仮想アドレスに相当するTLBのエントリを無効化する手法を採る．このためのペナルティは5クロックという．この数値が大きいか小さいかはキャッシュミスの頻度によるが，仮想キャッシュが高速化のために有効ということになれば，今後もシノニム解決のためのいろいろな手法が生まれてくるであろう．

また，命令キャッシュに関しては，性質的にメモリ内容の変更をともなわないため，アクセスタイムが有利な仮想アドレスキャッシュが採用されるケースが増えているようである．つまり，ライトバックをしないので，仮想アドレスに対応する物理アドレスが何であろうとあまり関係ない．キャッシュタグにプロセスIDを付加するか，タスク切り替え時に全エントリを無効化することで，ほとんどの場合事足りる．

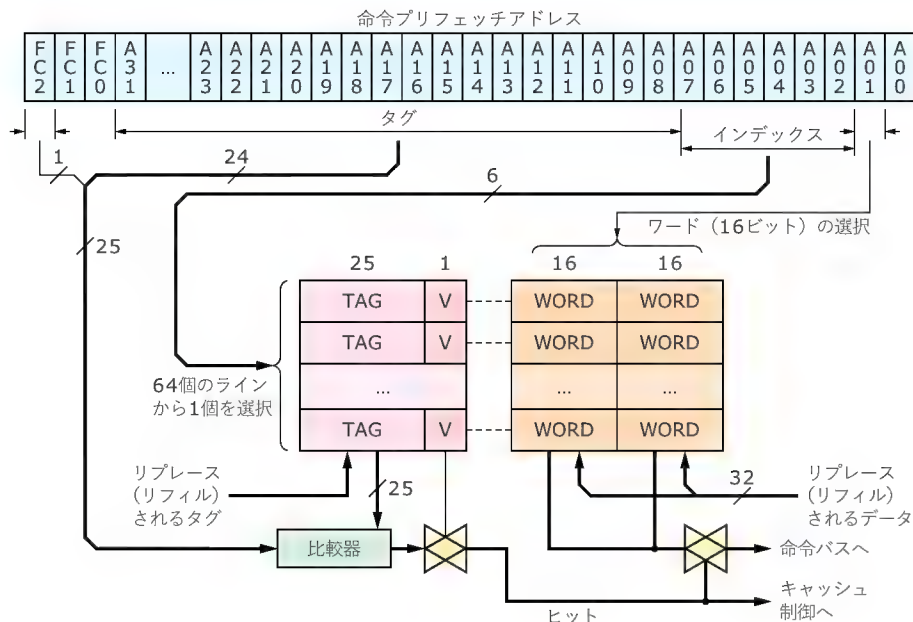
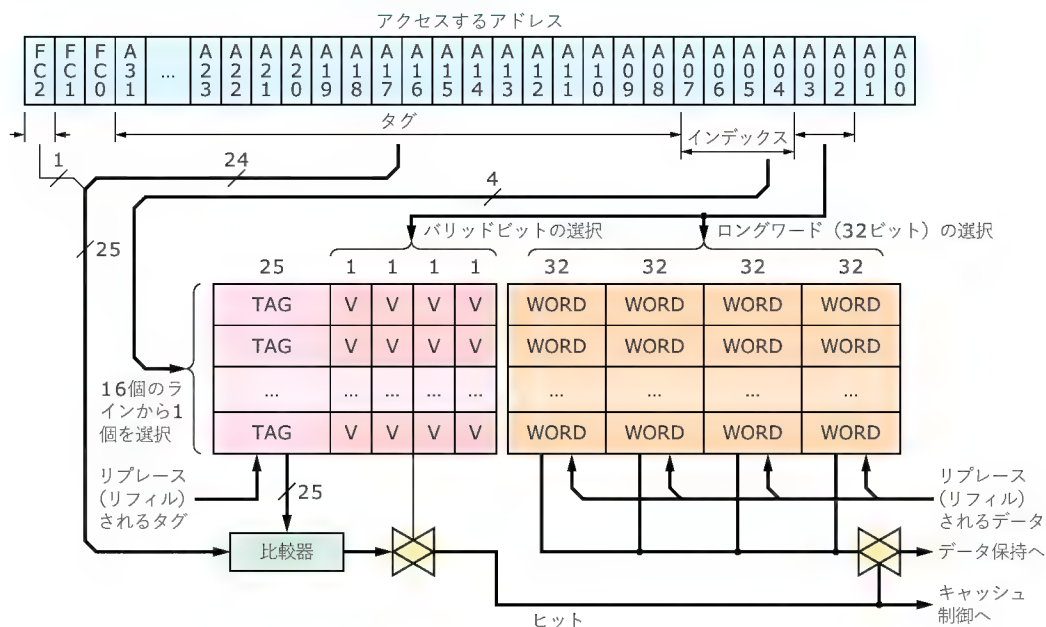
AMR10までのARMプロセッサやSH-5のほかにも，比較的新しいところでは，MIPSのRuby（R20K）が命令キャッシュに採用された．データキャッシュは，MIPSの従来方式である，仮想インデックス/物理タグである．

6 実際のプロセッサのキャッシュ構成

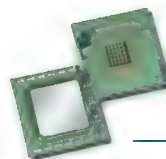
● MC680x0でのキャッシュ構成

MC680x0ではMC68020でキャッシュが内蔵された．ただし，命令キャッシュのみである．その構成を**図15**に示す．256バイトのダイレクトマップ方式で，1ラインは1ワード（4バイト）の容量をもつ物理アドレスキャッシュである．タグ部には機能コードのビット2（ユーザー/スーパバイザの表示）も含まれ，タグの比較時にアドレスと同時に比較される．MC680x0では同一の物理アドレスでも機能コードによって物理空間が区別されるからである．

MC68030では命令キャッシュに加えて，データキャッシュも内蔵された．**図16**にMC68030のデータキャッシュの構成を示す．256バイトのダイレクトマップ方式で，1ラインは4ワード（16バイト）の容量をもつ物理アドレスキャッシュである．書き込み制御はライトアロケート可能なライトスルー方式である．

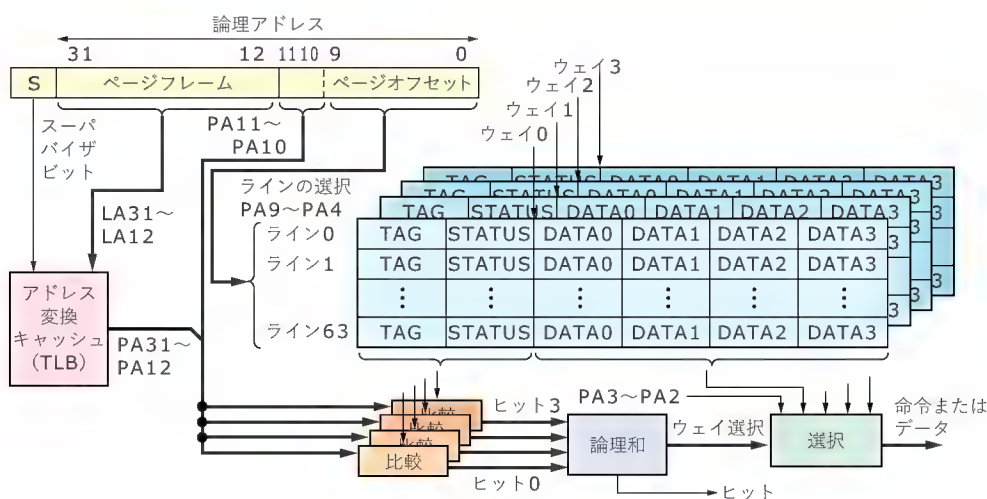
〔図15〕
MC68020の命令キャッシュ

〔図16〕
MC68030の命令キャッシュ

1ワードの容量がMC68020に比べ4倍に拡張されている．本来なら1ラインに1ビットあれば十分なバリッドビットがワードごとに用意され，全部で4ビットあるのが特徴である．キャッシュのリフィルを1ワード単位でも4ワード単位（バースト転送）でも行えるような設定が可能なためであろう．なお，命令キャッシュもまったく同じ構成をしている．**図17**にMC68030のキャッシュ制御レジスタ（CACR）を示す．この図を見ればわかるが，キャッシュロック（凍結）も可能である．

MC68040ではキャッシュ構成ががらりと変更された．**図18**にキャッシュの構成を示す．4Kバイトの4ウェイセットアソシアティブ方式で，1ラインは4ワードの容量をもつ仮想インデックス物理タグキャッシュである．書き込み制御はMMUでページ単位にライトスルー（ライトアロケートはしない）方式とライトバック方式を選択できる．リプレースメント方式はランダムである．

図19に命令キャッシュの，**図20**にデータキャッシュのライン構成を示す．バリッドビットはラインに1ビットのみとなった．不思議なのは**図19**でワード単位にダーティビットが用意されている点である．リフィルやライトバックはライン単位に行う（バリッドビットが1ビットしかないため）のでラインごとに1ビットあれば十分なはずなのだが，おそらく，ライトのバスサイクルを減らすために，真にダーティなワードのみをライトバックするためなのだろう．これにより，ライトバスサイクルの節約になる．

〔図17〕MC68030 のキャッシュ制御レジスタ（CACR）

31〜14	13	12	11	10	9	8	7〜5	4	3	2	1	0
000000000000000000	WA	DBE	CD	CED	FD	ED	000	IBE	CI	CEI	FI	EI

WA：ライトアロケート
DBE：データバーストイネーブル
CD：データキャッシュのクリア
CED：データキャッシュの中のエントリのクリア
FD：データキャッシュの凍結
ED：データキャッシュのイネーブル
IBE：命令バーストイネーブル
CI：命令キャッシュのクリア
CEI：命令キャッシュの中のエントリのクリア
FI：命令キャッシュの凍結
EI：命令キャッシュのイネーブル

〔図18〕MC68040 の命令キャッシュ

〔図19〕MC68040 の命令キャッシュのライン構成

TAG	V	LW3	LW2	LW1	LW0

TAG：22ビットの物理アドレス情報
V：バリッドビット
LW*n*：32ビットのデータエントリ

〔図20〕MC68040 のデータキャッシュのライン構成

TAG	V	LW3	D3	LW2	D2	LW1	D1	LW0	D0

TAG：22ビットの物理アドレス情報
V：バリッドビット
LW*n*：32ビットのデータエントリ
D*n*：LW*n*に対応するダーティビット
INVALID：not V
VALID(Clean)：V and(not D0)and(not D1)and(not D2)and(not D3)
DIRTY：V and(D0 or D1 or D2 or D3)

〔図21〕MC68040 のキャッシュ制御レジスタ

31	30〜16	15	14〜0
DE	000000000000000	IE	000000000000000

DE：データキャッシュのイネーブル
IE：命令キャッシュのイネーブル

図21 に MC68040 のキャッシュ制御レジスタを示す．それぞれのキャッシュのイネーブル（許可）ビットしかなく，キャッシュロック機能はなくなった模様である．

● i486 のキャッシュ構成

i486 のキャッシュは，8K バイトの容量をもつ 4 ウェイセットアソシアティブ構成の物理アドレスキャッシュである．**図22** に i486 のデータキャッシュのブロック図を示す．データキャッシュの書き込み制御はライトスルーで，リプレースは疑似 LRU で行う．また，ライトアロケートは行わない．すなわち，リードミスでのみキャッシュをリフィルし，ライトミスではキャッシュをリフィルしない．

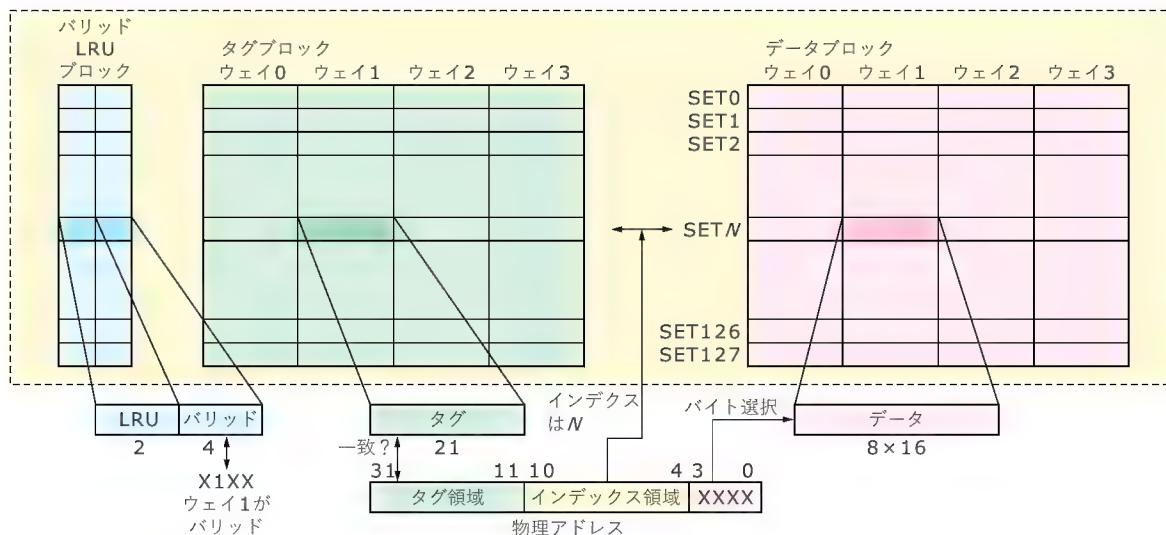
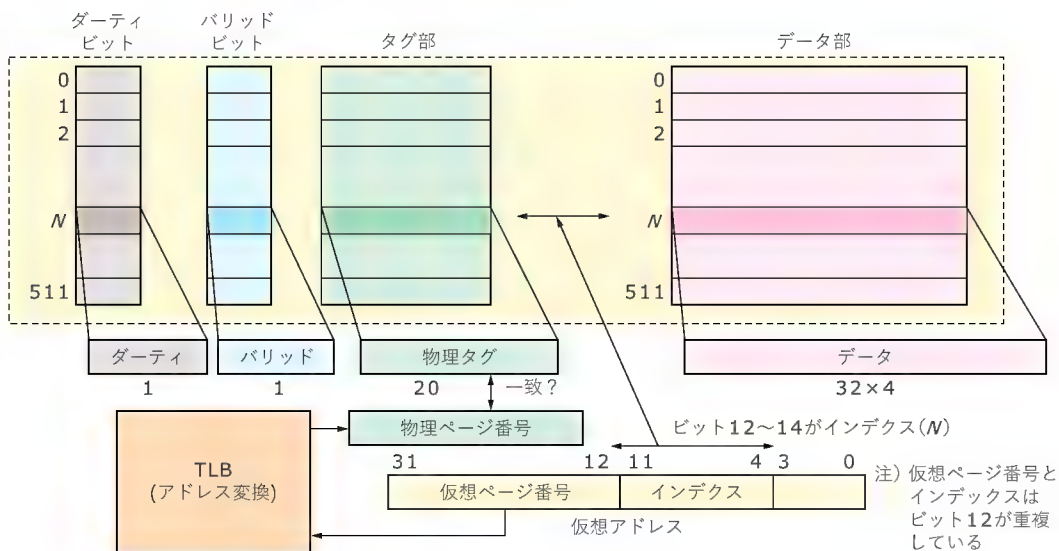
i486 のキャッシュにはバススヌープ機能がある．プロセッサバスにキャッシュラインインバリデーションが発生すると，アドレスバスが示すアドレスに一致するエントリを無効化する．

● R4000 のキャッシュ構成

MIPS R4000 のキャッシュは，8K バイトの容量をもつダイレクトマップ構成の仮想アドレスインデックス，物理タグキャッシュである．**図23** に R4000 のデータキャッシュのブロック図を示す．書き込み制御はライトバック方式で，リードミスまたはライトミスでキャッシュラインをリフィルする．キャッシュミス発生時，リフィルされるエントリのダーティビットが 1 なら，リフィル前に，古いキャッシュラインをメモリまたは 2 次キャッシュにライトバックを行う．

R4000 にもキャッシュのスヌープ機構がある．アドレスを指定して無効化を行うインバリデートプロトコルと，ラインの内容を更新するアップデートプロトコル（これは R4000MC/R4400MC のみ）がある．

MIPS 系のプロセッサはダイレクトマップ方式を採用していることが多い．2 ウェイセットアソシアティブキャッシュはハイ

〔図22〕
i486のデータキャッシュ構成

〔図23〕
R4000のデータキャッシュ構成(ページサイズ4Kバイトの場合)

エンドのR5000やR10000でしか採用されていなかった．最近では2ウェイセットアソシアティブ方式のものが増えてきているが，4ウェイはまだ珍しい．最近ではRuby(R20K)が4ウェイセットアソシアティブを採用しているのみである．ただし，MIPS社が提供するIPコアであるJade(4Kc)やOpal(5Kc)は1ウェイ(ダイレクトマップ)から4ウェイまでの構成を選択できるようになっている．とはいえ4ウェイ構成では消費電力が多くなるので，ウェイ予測などを行って電力を削減する工夫をしないと，組み込み用途には向かない．

ちなみに，Rubyはウェイ予測を行っている．また，最新のIPコアである24K(Topaz)は性能重視で4ウェイ構成のみになった．MIPS R4000のキャッシュを4ウェイアソシアティブ方式にすると，MC68040のキャッシュ構造に近くなる．

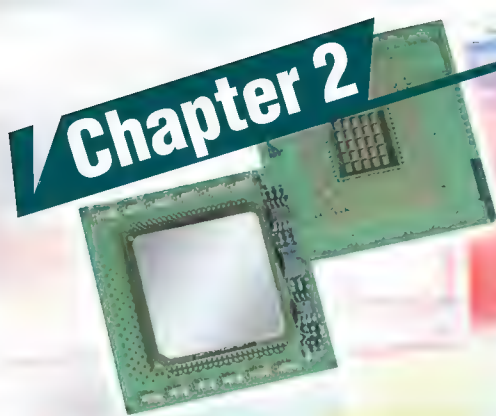
まとめ

主として，MPUに内蔵されているキャッシュの概要を述べてきた．キャッシュの動作を少しでも理解していただければ幸いである．なお今回は，マルチプロセッサ構成時のキャッシュの動作については複雑になるので意図的に省いている．

ところで，本稿では*n*ウェイセットアソシアティブにおける*n*個のダイレクトマップ形式のキャッシュを指すものとしてウェイという表現を使ってきたが，本来の意味が「*n*通りのセット」ということを考えると「セット」といったほうが正確かもしれない．まあ，ウェイと表現するのは筆者の職業病(?!)なので勘弁願いたい．また，ダイレクトマップという表現も正確にはダイレクトマップトである．

ところで，最近ではキャッシュのことをCASH(現金)との洒落で$と記述することも多い(¥でないところが米国発祥の洒落であることを感じさせる)．たとえば，命令キャッシュやデータキャッシュは，それぞれ，I$，D$と略記されることもあるので覚えておこう．

なかもり・あきら　フリーライタ

Chapter 2

仮想記憶/メモリ保護機能を実現するために

MMUの基礎と実際

中森　章

ここではWindowsやLinuxなど，仮想記憶を使う場合に必須となるMMUについて解説する．通常は仮想記憶を使わないことの多い組み込み用途であっても，信頼性の高いシステムを構築するためにMMUのメモリ保護機能を使う場合もある．ここでは，アドレス変換，TLB（Translation Look-aside Buffer），PTE（Page Table Entry），メモリ保護機能について解説したあと，680x0系やx86系，MIPSやPowerPCのMMUについて解説する．（編集部）

はじめに

MMUとはMemory Management Unitの略語である．つまり，メモリ管理ユニットのことで，MPUの外部または内部にあって仮想記憶機能を実現する．単にC言語などでプログラミングするだけなら，仮想記憶の知識などはほとんど必要ない．しかし，プログラムのサイズが一昔前に比べてはるかに巨大化しており，またマルチタスクが当然のように行われている昨今，その裏方にはMMUという働き者がいることを心に留めておいてほしい．

1 仮想記憶とは

● 仮想的に広大なメモリを用意する

その昔，まだメモリが高価だった頃，コンピュータに実装できるメモリ容量はわずかなものだった．時としてアプリケーションプログラムの容量は実際の物理メモリの容量を超え，そのようなプログラムを動作させるためにはアプリケーションプログラム側で細工をする必要があった．

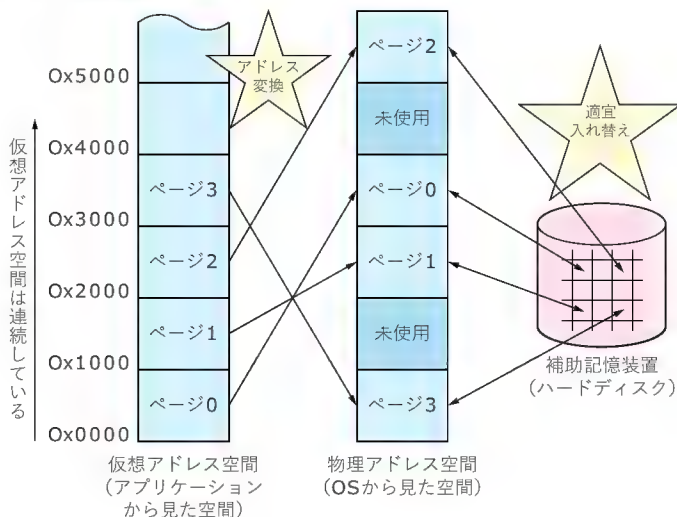
プログラムの性質として見ると，ある瞬間瞬間に実行されているのは全体の一部分にすぎない．そこで，プログラムをいくつかのブロックに分割し，必要な部分だけをメモリにロードして実行させ，不要になったらそのブロックを補助記憶装置（多くの場合，ハードディスク）に退避し，代わりにほかの必要なブロックを補助記憶装置から取り出して，新しいブロックと入れ替えるしくみが必要になる（**図1**）．しかし，実装されている物理メモリの容量を考慮しながらプログラミングをするのは効率的でないし，物理メモリの容量が変化すると，同じプログラムが使用できなくなってしまう．

そこで，このようなメモリ管理をOSにまかせるしくみが考案された．これが仮想記憶の原点である．仮想記憶を利用すると，ユーザーは物理メモリを意識することなく，物理メモリの容量を超えるような巨大なプログラムを実行できる．

● マルチタスクを実現する

PCはもとより，現在では規模の大きな組み込み機器は，そのほとんどがマルチタスクで動作している．マルチタスクとは複数のタスク（プログラム）を同時に物理メモリに置き，ある決められた順番に少しずつ（その多くは時分割で）実行していくものである．この場合，各タスクが必要とする全部の領域を物理メモリに割り当てようとすると，メモリに入りきらなくなってしまう．物理的に限られた容量しかないメモリを，多くのタスク間で分割して使用する手段が必要である．この場合も仮想記憶が有効である．そのため，仮想記憶といえば，現在ではマルチタスクを実現する手法として紹介されることが多い．

マルチタスクも，物理メモリを複数のブロックに分けて，そのブロックを各タスクに割り当てて実行させることで実現される．このような仮想記憶を行う場合，各タスクが自身に割り当てられた物理メモリのブロック以外をアクセスしないように保護する機能も必要になってくる．

〔図1〕仮想記憶のイメージ

● 現在ではページング方式が主流になっている

仮想記憶の方式としては，大きく分けて，**セグメント方式**と**ページング方式**がある．現在ではページング方式が主流なので，ここではページング方式を主体に話を進める．セグメント方式についてはコラムで言及する．

ページング方式の場合，タスクのアドレス空間を分割したブロックを**ページ**と呼ぶ．また，不要になったページを補助記憶装置に退避したり，必要なページを補助記憶装置から復元する作業を**ページスワップ**と呼ぶ．

メモリのアクセス速度にくらべてハードディスクのアクセス速度は非常に遅いので，ページスワップが頻繁に発生すると，プログラムの実行速度は低下する．しかし，プログラムとデータにはある程度局所性があるため，ページスワップがあまり発生しないことを期待して仮想記憶が実現されている．ところが，頻繁にページの範囲を超えて分岐が発生するプログラムや不連続な大量のデータを参照するプログラムでは，ページスワップの発生する確率が高くなる．このような場合は，そのタスクのページサイズを大きくすることで，ある程度ページスワップを回避できる．このため，MPUによってはタスクごとにページサイズを可変にできるようになっている．

2 アドレス変換

● アドレス変換とは？

PCにおけるプログラミングにおいて「このプログラムは物理アドレスの何番地に割り当てられるから」などと考えてプログラムを作る人は（OS屋などを除き），まずいない．誰もが，自分の書いたプログラムは，たとえば「0番地から配置され無限の容量をもっている」と考える．つまり，プログラムはそれぞれ固有のアドレス空間をもっている．マルチタスクを行うということは，重複するアドレス空間をもつ複数のプログラム（タスク）を同時に物理メモリに割り当てて実行するということである．このような操作を可能にするためには，プログラムの中で想定されているアドレスを，実際の物理メモリに配置するためのアドレスに読み替えるしくみが必要になる．これが**アドレス変換**である．

プログラムが想定しているアドレスは**仮想アドレス**（論理アドレスともいう）と呼ばれ，物理メモリに割り当てられるアドレスを**物理アドレス**（実アドレスともいう）と呼ぶ．アドレス変換とは，仮想アドレスを物理アドレスに変換する作業のことである．プログラムの仮想アドレス空間は一定の容量をもつページに分割される．このページ単位に，仮想アドレスから物理アドレスの変換が行われる（**図1**）．

ページのサイズ（容量）はOSによってまちまちである．昔は1ページのサイズが2Kバイトのものが多かったが，現在は4Kバイトのものが多いようだ．4Kバイトは16進数で表現すれば1000バイトである．私見ではあるが，人間にとってなんとなくきりのいい数値なので，OS屋さんに好まれるのであろう．

それはともかく，仮想アドレスと物理アドレスの対応は，物理メモリ上に置かれたアドレス変換テーブルによる．この変換テーブルはページテーブルと呼ばれ，通常4バイトまたは8バイト長のエントリの集まりである．これをとくにページテーブルエントリ（PTE）と呼ぶ．32ビットOSの場合，アドレスは32ビットで表現されるので，PTEには最低でも1ページあたり32ビット（4バイト）の領域が必要である．

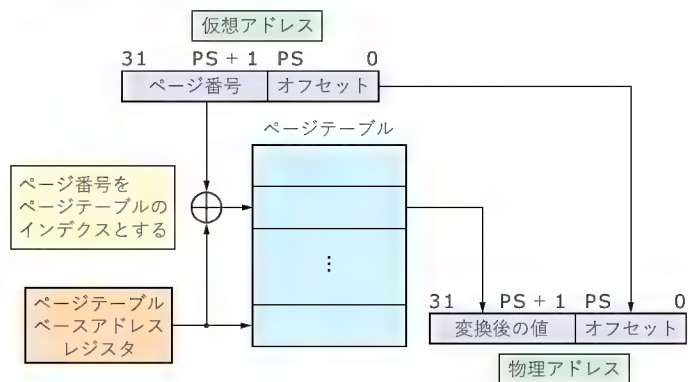
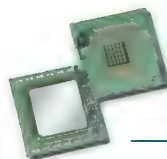
もっとも，仮想アドレスと物理アドレスは同一ページ内のオフセット（1ページが4Kバイトの場合はアドレスの下位12ビット）は一致するので，必要なビット数はもう少し少なくてよい．しかし実際には，そのページの保護情報のための情報やページスワップのための情報も必要になるし，ワード長（4バイト）またはダブルワード長（8バイト）のほうが（OSの）プログラムで扱いやすいので，一つの仮想アドレスに対して4バイトまたは8バイトのPTEが用いられるのが普通である．

● アドレス変換のレベル

さて，仮想アドレスが32ビット，1ページが4Kバイトの場合を考えよう．この場合，仮想アドレスの下位12ビットがページ内オフセット，上位20ビットがページ番号になる．このページ番号をインデックスとしてページテーブルを参照すれば，そのページに対応する物理アドレスを取り出すことができる．

なお，ページテーブルのベースアドレスはタスクごとに固有な値をもっていて，コンテキスト（タスクを性格づける情報）の一部である特権レジスタに格納されている．**図2**に仮想アドレスから物理アドレスを得る変換作業の概念図を示す．この図では20ビットのインデックスでページテーブルを参照するので，PTEの数は1M個必要である．PTEの容量は4バイトまたは8バイトなので，1タスクあたり4Mバイトまたは8Mバイトの物理メモリの容量がページテーブルのために必要になる．

しかし，タスクのもつアドレス空間は32ビット（4Gバイト）のすべての領域を被っているわけではなく，命令，データ，スタックなど，性質の異なる領域ごとにある程度塊になって存在している．このような場合，1M個のページテーブルエントリ

〔図2〕1レベルのアドレス変換

をすべて用意するのは不経済である．へたをしたら物理メモリがページテーブルだけであふれてしまうという状況も起こりかねない．そこで，ページテーブルを多段階に分けて参照する方法が考えられている．

この方式では，仮想アドレスのページ番号をさらにいくつかの領域に分ける．たとえば，20ビットのページ番号を上位12ビットと下位8ビットに分ける．この場合，上位12ビットをインデックスとして1段目のテーブルを参照し，2段目のテーブル(これがページテーブル)へのベースアドレスを獲得する．そして，下位8ビットをインデックスとして2段目のテーブルを参照し，物理アドレスを獲得する．この概念図を**図3**に示す．**図2**ではページテーブルを直接参照しているので1レベルのページング，**図3**では2回目でページテーブルを参照しているので2レベルのページングと呼ばれる．

〔**図3**〕**2レベルのアドレス変換**

最近のMPUでは，2レベルのページングでアドレス変換を行うことが主流であるが，MC68030や68040では3レベルのページングを行うこともできる．

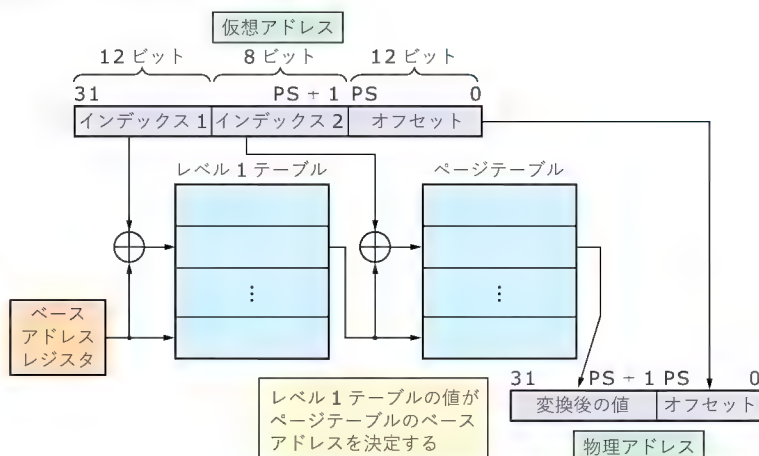
3 TLB

● TLBとは？

MPUが仮想記憶モードで動作している場合，仮想アドレスから物理アドレスへの変換を，いちいち物理メモリ上のページテーブルを参照しにいっていたのでは，その処理が命令実行のボトルネックになってしまう．それを避けるために，MPUは内部に**TLB**(Translation Look-aside Buffer)と呼ばれる変換テーブルをもっている．

日本語ではアドレス変換緩衝機構と訳されることが多い．モトローラは**ATC**(Address Translation Cache)，つまり，アドレス変換キャッシュと呼んでいる．その名のとおり，TLBとは，PTEをチップ内にキャッシュしたものである．

MPUはアドレス変換を行うとき，まずTLBを参照し，そこに目的の仮想アドレスと物理アドレスのペアが格納されていれば(TLBヒット)，その物理アドレスを用いて命令を処理する．もし該当する仮想アドレスがTLB内になければ(TLBミス)，物理メモリ上のページテーブルを参照しに行き，その値をTLBに登録する．また，TLBにはPTEと同様にメモリ保護などの情報が格納されており，TLB参照の際に不正アクセスがないかチェックする．もし不正なアクセスである場合は，メモリ保護例外を発生する．以上がMMUの機能である．

ただし，最近のRISCチップでは，TLBを参照したとき，仮想アドレスが登録されていないとただちに例外を発生して，TLBの内容を入れ替える処理をOSのプログラムにまかせる．何度もメモリ上のテーブルを参照してTLBの内容を更新する処理は，実現が複雑であり，メモリアクセスはロード/ストア命令だけというRISCのポリシにも反する．何よりもパイプライン動作が妨げられてしまう．このためRISCでは，TLBの機能そのものがMMUの機能ということもできる．

● TLBの構造(連想方式)

TLBとは仮想アドレスをタグとして内容を参照し，一致するタグがあれば対応するデータを物理アドレスとして出力する一種のキャッシュメモリである．その構造は参照の仕方により，次の3種類に分類できる．

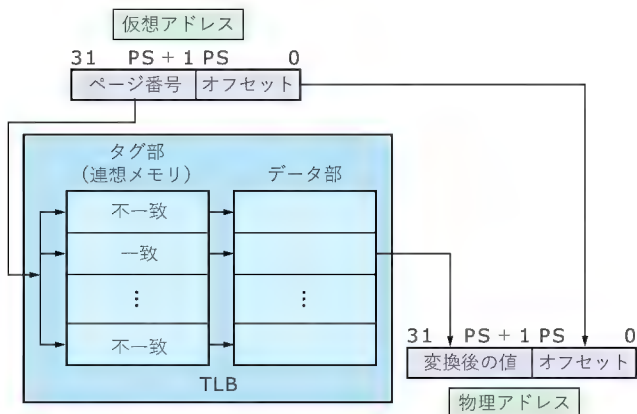
- フルアソシアティブ方式
- ダイレクトマップ方式
- Nウェイセットアソシアティブ方式($N \geqq 2$)

▶フルアソシアティブ方式

フルアソシアティブ(Full Associative)方式は，TLBのエントリ数の数だけ異なる仮想アドレスを格納できる方式である．ほかの方式とは異なり，各エントリに格納される仮想アドレスに制限はない．通常は連想メモリという特殊なメモリで構成されるため，LRU処理(詳細は後述)が複雑になるため，多くのエントリをもたせることができない．現在の技術では50エントリ程度が限界と思われる．ただし，実装されているエントリをむだなく使用することができるので，少ないエントリ数でも高いヒット率(仮想アドレスを参照したとき，TLB内に存在する確率)を得ることができる．**図4**にフルアソシアティブ方式のTLBの構成を示す．

▶ダイレクトマップ方式

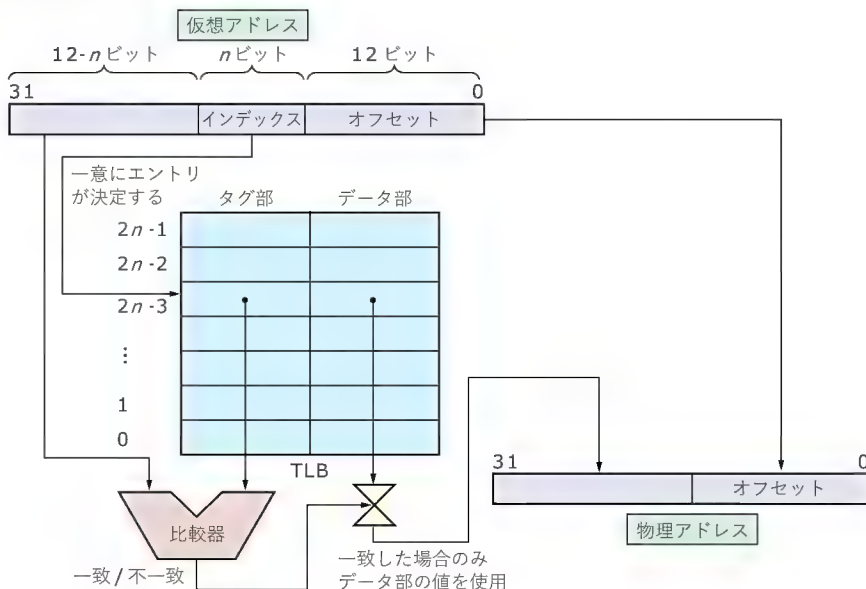
ダイレクトマップ(Direct Mapped)方式とは，もっとも単純な方式である．仮想アドレスが決まると，その仮想アドレスで参照するエントリが一意に決まってしまう．たとえば，256エントリのダイレクトマップ方式のTLBを参照する方法として，仮想アドレスのビット19～12(8ビット)を使用してエントリをインデックスする方法が考えられる．これは，ページサイズが4Kバイトの場合である．仮想アドレスのビット31～12がページ番号を表し，その下位8ビットである．

〔図4〕フルアソシアティブ方式

8ビットのデータは256種類を識別できるので，仮想アドレスとTLBのエントリを1対1に対応させることができる．

ただし，この場合，下位8ビットが一致する仮想アドレスは異なるアドレスであっても同一のTLBエントリが参照されてしまう．プログラムの仮想アドレスが256通りでまんべんなく変化することは希なので，場合によっては一度も参照されないエントリが存在する．逆に同じエントリが何度も参照され，前のデータを書き潰してしまう恐れもある．

ダイレクトマップ方式は，構造は単純でエントリ数を多くもたせることができるが，エントリ数を多くしないと高いヒット率は期待できない．**図5**にダイレクトマップ方式のTLBの構成を示す．

▶ ***N*ウェイセットアソシアティブ方式**

***N*ウェイセットアソシアティブ**(N-way Set Associative)方式とは，ダイレクトマップ方式の改良版である．ダイレクトマップ方式のエントリをN系統用いて構成する．この方式も，LRU処理の制限からNの値は2または4であることが多い．簡単のために2ウェイセットアソシアティブ方式の場合で説明する．ダイレクトマップの場合と同様に，仮想アドレスが与えられるとエントリは一意に決定されるが，今の場合は2組のウェイ(エントリの集合)があるので，同時に二つのエントリに格納されている仮想アドレスと比較を行う．与えられた仮想アドレスがそのどちらかに一致していればヒットということになる．一般にウェイ数が増える程ヒット率が向上する．**図6**に2ウェイセットアソシアティブ方式のTLBの構成を示す．

● TLBの更新方式

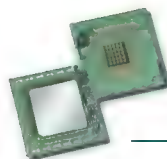
TLBのエントリ数には限りがある．エントリの中に有効なデータが入ってなければ，そこにアドレス変換の情報を格納していけばよいが，エントリがすでに有効なデータで占められていて，新たに変換の情報を登録できないことがある．この場合は，古い情報を追い出して新しい情報を書き込む(上書きする)ことになる．

追い出しの対象となるエントリを決定するためにもっとも多く使われるのが，LRU(Least Recently Used)という手法である．つまり，時間的にもっとも使用されていないエントリを追い出す．その実現方法は2ウェイセットアソシアティブ方式では簡単である．二つのエントリの組に対して1ビットのLRUビットを設ける．そのビットの値が0か1によって二つのうち対応するエントリをあらかじめ決めておく．そして，0側のウェイがヒットすればLRUを1側に，1側のウェイがヒットすればLRUを0側に更新する．もし，そのエントリに対応する仮想アドレスであって，どちらのエントリの内容とも一致しない仮想アドレスを変換しなければならない場合は，対応する物理アドレスを求め，LRUビットが示す側のウェイのエントリに上書きする．4ウェイセットアソシアティブの場合は四つのエントリに対して6ビットの情報でLRUを構成できる．フルアソシアティブ方式でのLRUはかなり複雑である．その方式が特許になるほどややこしいので，ここでは説明を省略する．

現実でも，エントリ数が多いTLBに対してはLRU方式を用いない．それでは，フルアソシアティブ方式の場合，追い出すエントリをどのように決定するのか．答は単純である．適当に決めるのである．具体的には(疑似)乱数を用いてエントリを決定する．これは，どのエントリの仮想アドレスも同じ程度に参照されていると仮定している．どのエントリが選ばれても恨みっこなしということである．

● タスク切り替えとTLB

タスクの仮想アドレス空間はタスクごと

〔図5〕ダイレクトマップ方式

に固有である．意図的にほかのタスクのアドレス空間と一部分の空間を共有させることもあるが，基本的には特定の特権レジスタの値で一意に規定される．この特権レジスタはコンテキストの一部であり，その値を基準として何回か間接参照を繰り返せば，最終的にページテーブルのベースアドレスを得ることができる．このため，タスクが切り替われば TLB の内容も，そのタスクの仮想アドレス空間を反映したものに切り替わらなければならない．論理的にはタスクの数だけ TLB が必要ということになる．しかし，現実的には，タスクの数の最大値を予測することは不可能であり，MMU内にいくつもの TLB を実装するのはむだが多い(実質不可能)．

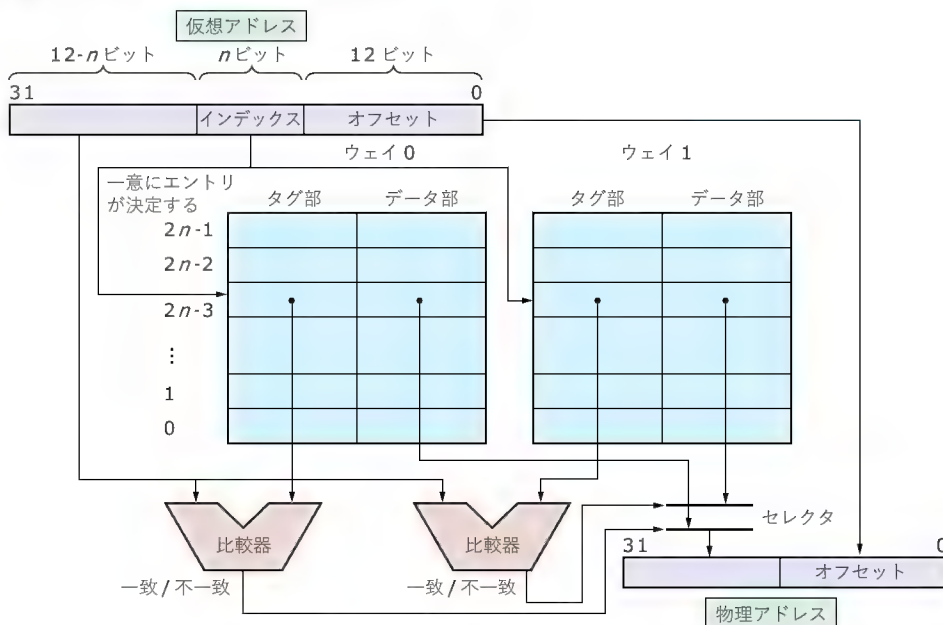
〔図6〕2ウェイセットアソシアティブ方式

そこで，多くの MPU ではタスクが切り替わるたびに TLB の内容を無効化してしまう．この方式では，必要以上に TLB エントリを無効化してしまうおそれがあり，それがプログラムの実行速度の低下を招く．

たとえば，タスク番号0のタスクでは仮想アドレス1000番地しか参照せず，またタスク番号1のタスクでは仮想アドレス2000番地しか参照しない場合で，タスクが0→1→0と切り替わる場合を考える．このとき，最初は1000番地が TLB に登録されているが，タスクが1に切り替わる時点で無効化される．そして，タスクが再び0に切り替わるとき，1000番地は TLB に登録されてないので，再びメモリ上のアドレス変換テーブルを参照して TLB に1000番地を登録する必要がある．タスク1が1000番地を使用しないなら，この TLB 入れ替え処理は余分である．しかし，他のタスクが使用する仮想アドレスを予測することはできないので，誤ったアドレス変換をしないように，古いタスクのアドレス変換情報は消去してしまわなければならない．必然的に，しなくてもよい TLB 入れ替えが増加する．

その欠点を回避するために，TLB のタグ部にタスク番号を入れておき，タスク番号込みで仮想アドレスの一致を調べるという方式を採用する MPU もある．この方式だと，タスク番号1の仮想アドレス0番地と，タスク番号2の仮想アドレス0番地が同時に TLB に登録されていても(このような状況が発生するのはフルアソシアティブ方式の TLB に限られるが)，二つの0番地を区別することができる．タスクの切り替え時に TLB の内容を無効化する必要もない．TLB 入れ替えは本当に必要な場合にのみ行われる．

● TLBの分離

最近の MPU はパイプライン処理で命令を実行している．命令フェッチやデータアクセスの前には仮想アドレスを物理アドレスに変換する必要があり，そのとき TLB が参照される．何も考えずに MPU を設計すると，ある瞬間に，命令用のアドレス変換での TLB の参照と，データ用のアドレス変換での TLB の参照が同時に発生してしまう．命令の仮想アドレスとデータの仮想アドレスは一般には一致しないので，二つの仮想アドレスで同時に TLB を参照することになるが，これは不可能である．どちらかの参照を遅れさせて，逐次的に参照をすることになる．

このときのパイプラインの乱れを嫌って，命令用とデータ用に二つの TLB を採用する MPU もある．キャッシュで命令とデータのデータパスをそれぞれ専用にもたせる構造をハーバードアーキテクチャと呼ぶが，それの TLB 版と考えればよいだろう．実際，古くからハーバードアーキテクチャを提唱していたのはモトローラであり，モトローラの68040などは，命令とデータの2系統の TLB をサポートしている．

● マイクロTLB

命令とデータで同じ規模の TLB を用意するのは大げさだし，あまり効果はないように思える．なぜなら，データはともかく，命令のアドレスはシーケンシャルに実行され，分岐で初めて別のアドレスに切り替わるからである．分岐自身もページサイズの範囲を超えることは希なので，命令のための仮想アドレスを変換しなければならない場合は(データに比べると)極端に少ない．そこで，命令用の TLB として1～4エントリ程度の特別な TLB を採用する MPU もある．そのような TLB は**マイクロTLB**と呼ばれる．

多くの場合，マイクロ TLB は本体の TLB の内容をキャッシュしたもので，ページサイズも固定である．命令がマイクロ TLB

にミスした場合は，まず，本体のTLBを参照し，そこにヒットすれば，そこから物理アドレス情報をもってきて内容を更新する．TLBのページサイズがマイクロTLBのページサイズよりも大きい場合は，マイクロTLBでミスしても本体のTLBでヒットする確率が高いので，アドレス変換テーブル検索のためのメモリアクセスが発生することは希である．また，このような構成であれば，メモリアクセスを発生させてTLBを更新するロジックが1系統分で済む．一方，命令TLBとデータTLBに分離されている場合は，それぞれ独立なTLB更新ロジックが必要である．

さらに，マイクロTLBの参照は，本体の巨大なTLBを参照するよりも少ない電力で行えるので，命令だけでなく，データに対してもマイクロTLBが採用されることもある．

4 PTE（Page Table Entry）の実例

● PTEとは？

TLBミス時，アドレス変換におけるメモリ内の変換テーブルのサーチは2〜3段階のレベルを分けて行われる場合もあるが，最終的には，ページテーブルと呼ばれる**PTE**（Page Table Entry）が順次格納されているテーブルに突き当たる．PTEとは通常31ビット長のデータで，物理アドレス（オフセット部分を除く）と保護情報を含んでいる．PTEはページディスクリプタと呼ばれることもある．

ページテーブルに並んだPTEの意味は，先頭が仮想アドレス0（ページ0）に対応する情報，その次が仮想アドレス0x1000（ページ1，ページサイズが4Kバイトの場合）に対応する情報，その次が仮想アドレス0x2000（ページ2）に対応する情報，という具合になっている．何番目のPTEが使用されるかは仮想アドレスの値によって一意に決定される．**図7**にMC680x0で使用されるPTEとx86で使用されるPTEの実例を示す．PTEの情報のうち，物理アドレスに関しては説明不要と思うが，他のビットについて説明しておこう．

● MC680x0の場合

CM：キャッシュモード

このビットは対応する仮想ページの，キャッシュの可/不可，ライト制御（ライトスルー/ライトバック），キャッシュ不可時にアクセスの逐次性を保証するか否かを示す．

G：グローバル

このビットはPFLUSH命令で使用する．PFLUSHはTLBのエントリを無効化する命令であるが，Gビットがセットされているページは無効化されない．

M：モディファイ

ライトアクセスで発生したTLBミスに起因する変換テーブルのサーチが行われた後，対応するPTEのMビットが自動的にセットされる．つまり，対応する仮想ページの内容が変更されたことを示す．

PDT：ページディスクリプタタイプ

ページディスクリプタ（PTEを含む変換テーブルエントリ）の種類を示す．それは，有効/無効，対応するテーブルまたはページがメモリ内に存在/不在，間接（中間）のディスクリプタか否かという情報を示す．間接ディスクリプタの場合は次のレベルの変換テーブルの先頭を示す物理アドレスが格納されている．直接（最終）ディスクリプタの場合はそれがPTEであることを示す．なお，MC680x0では，PTEに対応する物理ページの内容がメモリに存在することをレジデントと呼ぶようである．

S：スーパバイザ保護

スーパバイザモードのみで参照できるページであることを示す．Sビットがセットされていない場合は，スーパバイザモードでもユーザーモードでも参照できる．

U：使用

変換テーブルのサーチが行われた後，対応するPTEのUビットが自動的にセットされる．Mビットとは異なり，リード，ライト両方のアクセスでセットされる．対応する仮想ページの内

〔図7〕PTEの実例

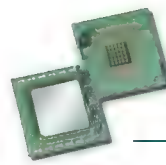
31–12	11	10	9	8	7	6–5	4	3	2	1–0
物理ページ番号	UR	G	U1	U0	S	CM	M	U	W	PDT

CM：キャッシュモード
G：グローバル
M：モディファイ（変更）
PDT：ページディスクリプタタイプ（存在，間接）
S：スーパバイザ保護
U：使用（参照）
U1：ユーザーページ属性1
U0：ユーザーページ属性0
UR：ユーザー用
W：ライト保護

（a）MC680x0のPTE（ページディスクリプタ）4Kバイトページ用

31–12	11–9	8–7	6	5	4	3	2	1	0
物理ページ番号	OR	IR	D	A	PCD	PWT	U/S	R/W	P

OR：OS用
IR：インテル予約
D：ダーティ（変更）
A：アクセス（参照）
P：プレゼント（存在）
PCD：ページキャッシュ禁止
PWT：ページライトスルー
U/S：ユーザー/スーパバイザ（保護）
R/W：リード/ライト（ライト保護）

（b）x86PTE

容が参照されたことを示す．

U0，U1：ユーザーページ属性

MPUの実行に影響は与えない．それぞれの値が，UPA0，UPA1という端子状態に反映される(MC68040以降)．

UR：ユーザー使用

ユーザー(OS)が自由に使用してよいビット．MPUにとっては意味がない．

W：ライト保護

このビットがセットされている仮想ページに対し，ライトアクセスを行おうとすると例外が発生する．

● x86の場合

OR：OS用

OSが自由に使用してよいビット．MPUにとっては意味がない．

IR：インテル予約

将来の拡張用にインテル(メーカー)によって予約されているビット．現状，MPUにとっては意味がない．

D：ダーティ

ライトアクセスで発生したTLBミスに起因する変換テーブルのサーチが行われた後，対応するPTEのMビットが自動的にセットされる．つまり，対応する仮想ページの内容が変更されたことを示す．

A：アクセス

変換テーブルのサーチが行われた後，対応するPTEのAビットが自動的にセットされる．Dビットとは異なり，リード，ライト両方のアクセスでセットされる．対応する仮想ページの内容が参照されたことを示す．

PCD：ページキャッシュ禁止

このビットがセットされているPTEに対応する仮想ページはキャッシュアクセスを行わない．

PWT：ページライトスルー

このビットがセットされていると外部キャッシュ(L2キャッシュ)をライトバック制御にする．

U/S：ユーザー/スーパバイザ

このビットがセットされていないと，特権レベル3(ユーザーモード)では対応する仮想ページをアクセスできない．

R/W：リード/ライト

このビットがセットされていないと，特権レベル3(ユーザーモード)では対応する仮想ページに対してライトアクセスできない．

P：プレゼント

このビットがセットされていれば，対応する物理ページの内容がメモリに存在することを示す．このビットがセットされていない場合，ページフォールト(例外)が発生する．

これまでの説明を見ればわかるが，PTE内の情報は，どのMPUでも似たり寄ったりである．似ているのは，物理アドレス，アクセスがあったこと示す情報，ライトがあったことを示す情報，存在を示す情報，保護情報などである．個人的にはx86での名称がしっくりくるので，そちらを使って，以下に，OSがそれらの情報をどう利用するかを説明する．

● ページフォールト時の処理

ページフォールトとはPTEにアクセスした場合に，そのPTEが無効だったり，対応するページの内容がメモリに存在しないときに発生する例外である．

ページフォールトの処理には，Pビット(PDT = 00)を利用する．Pビットが0ならば，仮想アドレスに対応するプログラムの内容がメモリ内に存在しないことを意味する．初期状態ではPTE内の物理アドレス情報は決定されていない．ページフォールトが発生すると，OSはメモリ内に空いている領域を見つけ，そこに新しいページの内容を補助記憶装置(ハードディスク)からロードする(これをページインという)．このとき，見つかった空き領域の物理アドレスがPTE内の物理アドレス情報となる．メモリがすべて他のページに占有されていて空き領域がない場合は，どこかのページを補助記憶装置に追い出して(これをページアウトという)そこを使用する．ページインとページアウトの操作を総称してページスワップ(交換)と呼ぶ．

● ページスワップ時の疑似LRU(Least Recently Used)制御

ページアウトを行う場合，もっとも使用頻度の低いページを追い出すのが効率的である．Aビット(Uビット)を利用して疑似LRU処理を行い使用頻度の低いページを決定する．そのために，OSは定期的に全PTEのAビットの状態をチェックする．そしてチェックが終わったら，ソフトウェアでAビットを強制的に0にクリアする．同時にそのPTEの内容がキャッシュされているTLBのエントリを無効化しておく．TLBにヒットする限り，PTEのアクセスが発生しないからである．こうしておけば，その後，同じ仮想アドレスに対するアクセスが発生すると再びAビットが1にセットされる．このような環境でAビットが1になる頻度を計数しておき，それがもっとも小さいページがアクセスのもっとも少ないページということになる．

なお，OSによっては，疑似LRU処理を行わず，単純なFIFO処理でページアウトするページを決定するものもある．これは，いちばん昔に変換したページから追い出していくというものである．あるいは，どのページを選択しても大差ないとして，ランダム処理で適当に追い出す候補を決める場合もあるかもしれない．

● ページスワップ時の補助記憶装置への無意味な書き戻しを制御

メモリ内に存在するページでも，そこに対して書き込みが行われていなければ，その内容は補助記憶装置に存在するもの(ページイン直前のもの)と同じである．つまり，そのページがページアウトの対象になっても補助記憶装置に書き戻す必要はない．補助記憶装置から読み込んだ新しいページの内容でメモリを書き潰してよい．これは処理時間の短縮につながる．この書き込み制御にはDビット(Mビット)を利用する．Dビットが

Column

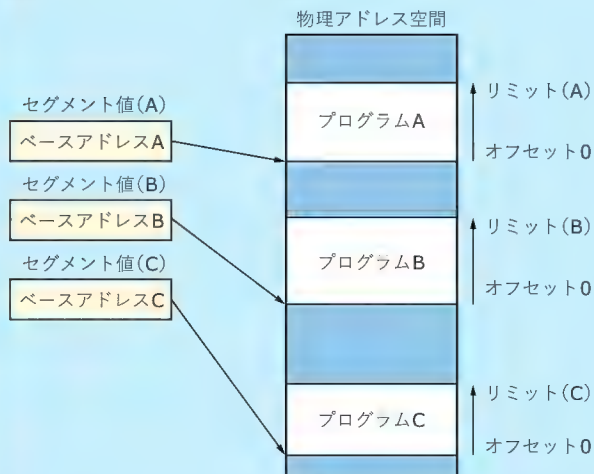
セグメント方式

● セグメンテーションの概念

本稿では仮想記憶の方式として，ページを単位とする，ページング方式を中心に説明してきた．ここで，もう一つの主要な仮想記憶方式であるセグメント方式について説明しておこう．これは，セグメントを単位とするのでセグメンテーションともいう．

セグメント方式ではアドレスの指定を，ベースアドレス(開始アドレス)とベースアドレスからのオフセット値で行う．そして，すべてのメインルーチンやサブルーチンといったモジュールは，オフセット0から開始され，データのアクセスもオフセットで指定するものと仮定する．こうすることで，そのモジュールはメモリ内のどこに配置しても実行可能になる．つまり，リロケータブル(再配置可能)となる．このため，モジュールごとに物理アドレスでベースアドレスを決定してやれば，同じオフセットを有する別のモジュールをメモリ内の自由な位置に置くことができる．モジュール自体は自身がメモリのどこに割り当てられるかを意識する必要はない(**図A**)．

セグメント方式では，仮想アドレスは，(物理アドレスで示される)ベースアドレスを直接/間接的に指定するセグメント値と，セグメント内のオフセット値という二つの情報で規定される．このため，セグメント方式のアドレスは2次元アドレスとも呼ばれる．一方，これまで述べてきた一つの情報で仮想アドレスを指定する方式のアドレスは，1次元アドレス，または線形アドレス(リニアアドレス)と呼ばれる．

〔図A〕セグメンテーションの概念

● セグメント単位でのスワップ

さて，大きなプログラムでは，コード部，データ部，スタック部が，それぞれいくつものセグメントに分かれている．このうち，ある時点のプログラムの実行に必要なセグメントのみをメモリに置いて実行すれば，物理メモリの容量を越えるプログラムを実行することもできる．これはメモリスワップの単位がセグメントになっただけで，ページングによる仮想アドレス方式と同じ効果を生む．

セグメント方式では，ページングでの変換テーブルに相当するものが，セグメントテーブルである．セグメントテーブルの各エントリは，メモリ保護情報，セグメント長(アドレスの上限)，ベースアドレスといった情報を含む．**図B**にセグメント方式でのアドレス変換を示す．MMUの挙動としては次のようになる．

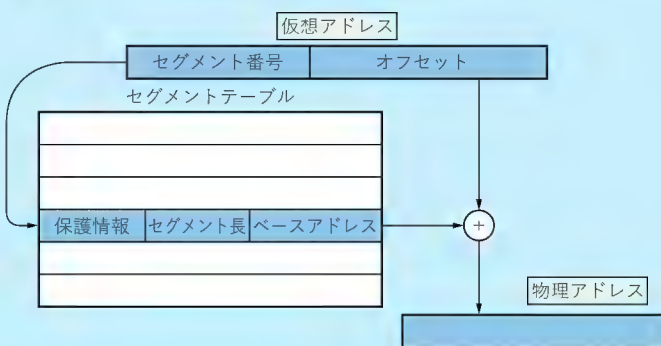
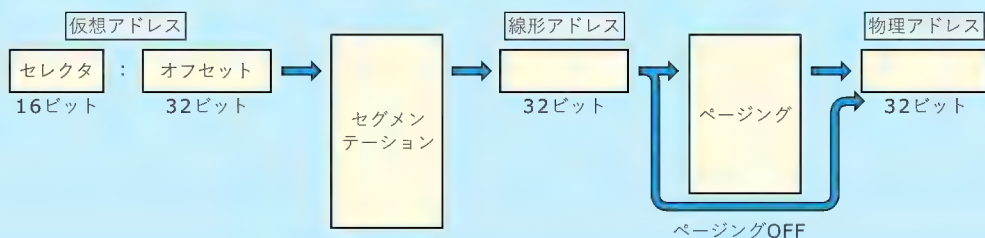
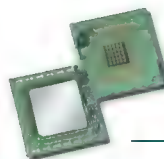
1) 仮想アドレスに含まれるセグメント値でセグメントテーブルをアクセスする
2) アクセスされたセグメントテーブルのエントリからベースアドレスを得る
3) ベースアドレスとセグメント内オフセットを結合して物理アドレスを生成する

x86アーキテクチャにおいて，セグメント値はセグメントレジスタに格納されているので，直接仮想アドレスの一部としては見えない．さらに，リアルモードにおいては，セグメント値を4ビット左シフトしてベースアドレスとしている．

● x86でのセグメンテーション例

x86アーキテクチャ(プロテクトモード)では，仮想記憶のアドレッシングにセグメンテーションとページングを併用している．**図C**に示すように，セレクタ値(セグメント値)とオフセットからなる2次元アドレスがセグメンテーションによって線形アドレスに変換され，それを仮想アドレスとしてページングを行って物理アドレスを得る．

セレクタとはセグメントテーブルへのインデックス，セグメント

〔図B〕セグメンテーションのアドレス変換

〔図C〕セグメンテーション＋ページング(x86)

〔図D〕セグメントレジスタ

INDEX：GDT/LDTへのインデックス
TI：テーブルインデックス
RPL：要求レベル

〔図E〕セグメントディスクリプタ

〔図F〕リニアアドレスから物理アドレスへの変換

テーブルの種類，要求特権レベルという三つの領域からなる16ビットの情報である．**図D**にセレクタを示す．x86では実行中のセグメントの保護レベルが現在の実行レベルになることはすでに述べたが，セレクタ中の要求特権レベルは実行レベルの特権性を下げる効果がある．つまり，現在の実行レベルと要求特権レベルを比較して特権性が低いレベルのほうが現在の実行レベルとみなされる．通常は，レベル0（最高の特権性）となっている．

- グローバルディスクリプタテーブルとローカルディスクリプタテーブル

セグメントテーブルにはグローバルディスクリプタテーブル（GDT）とローカルディスクリプタテーブル（LDT）の2種類がある．GDTとは，システム内に一つだけ存在するセグメントテーブルのことで，OSや複数のタスクから共通にアクセスされるメモリ領域を定義する．それに対し，LDTは，タスクごとのメモリ領域を定義する．そして，セレクタのインデックスは，GDTまたはLDT内のエントリ（それぞれをセグメントディスクリプタと呼ぶ）を選択する．テーブルインデックスは，このセグメントのディスクリプタがGDTであるかLDTであるかを示す．

GDT/LDTの各エントリは，セグメントディスクリプタと呼ばれる．これは，32ビットのベースアドレス，20ビットのリミット値，その他の情報から構成される64ビットのデータである．**図E**にセグメントディスクリプタを示す．セグメントディスクリプタのうち，G（Granularity）ビットはリミット値の単位を指定する．G＝0なら単位は1バイトであり，セグメントの大きさは0〜1Mバイトとなる．G＝1なら単位は4Kバイトであり，セグメントの大きさは0〜4Gバイトとなる．DPL（Descriptor Privilege Level）はそのセグメントの保護レベルである．DT（Descriptor Type）はセグメントディスクリプタの示すセグメントの種類（メモリセグメント，システムセグメント，ゲート）を指定する．メモリセグメントかシステムセグメント（またはゲート）かによって，セグメントディスクリプタのTYPE領域の意味が変わってくる．メモリセグメントではリード/ライト/実行の保護情報を指定する．システムセグメントではLDTまたはTSS（Task State Segment）という情報を指定する．ゲートではゲートの種類を指定する．

なお，メモリセグメントディスクリプタとシステムセグメントディスクリプタにはセグメントのベースアドレスが格納されている（オフセットは線形アドレスのオフセットと同じ）のに対し，ゲートディスクリプタにはポインタ（セレクタ値とオフセット値）が格納されている．ところで，セグメントディスクリプタのベースアドレスやゲートディスクリプタのオフセットが下位と上位に分離して格納されているのは80286との互換性のためである．

セグメンテーションの後はページングが行われるが，これは他のMPUと同様な機構なので詳細な説明は省略する．32ビットの仮想アドレスのビット22〜31をディレクトリという単位として1レベル目，ビット12〜21をページ単位として2レベル目のテーブルを引き，計2レベルのアドレス変換を行う（**図F**）．

0なら，そのページへの書き込みが行われていないことを示す．

● メモリマップトI/Oの実現

アドレス変換により，I/Oポートを仮想アドレスに対応させることもできる．その仮想アドレスをリード/ライトすることで，I/Oポートへのリード/ライトとみなすしくみを**メモリマップトI/O**という．これを実現する場合，そのページは非キャッシュ領域でなくてはならない．なぜなら，同じI/Oポートをリードしても同じ値が返ってくるとは限らないので，それがキャッシングされると都合が悪いからである．

PCDビット（CMビット）を使用すれば，そのページのキャッシングを禁止できるので，メモリマップトI/Oが実現できる．もっとも，PCDビットはI/Oポートでなく，フレームバッファなど，キャッシングされると都合の悪い領域の指定にも利用する．

また，I/Oポートはリード/ライトする順番が異なると意味が変わるので，アクセスの逐次化（プログラムで書いた順番にアクセスすること）を実現することも必要である．通常のMPUでは非キャッシュ領域に対するアクセスの逐次化は保証されているが，例外もある．たとえば，MC680x0では逐次化を明示的に指定する必要がある．

5 メモリ保護

● 実行レベルについて

悪意のあるアプリケーションプログラムが，あるいは，バグのあるアプリケーションプログラムの暴走で，OSの領域を壊さないように，MMUはメモリ保護の機能を提供する．上述のように仮想記憶モードではPTEによってリード/ライト属性による保護が実現される．通常，プログラム領域はリードのみ可，データ領域はリード/ライト可能に設定されている．このほかにもユーザー，カーネルといった特権性による保護が行われる．これについて説明しよう．

〔図8〕実行レベルとメモリ保護

MPUのアーキテクチャでは，プログラムの実行レベルというものが定義されている．これは特権性の強さを表すもので，通常，アプリケーションプログラムは最低の特権性の下で実行される．メモリ保護とは，仮想アドレス空間の各ページに保護レベル（そこのプログラムやデータにアクセスできる最低の実行レベル）をもたせ，特権性の低いプログラムから，より特権性の高いプログラムへのアクセスをできなくする機能である（**図8**）．

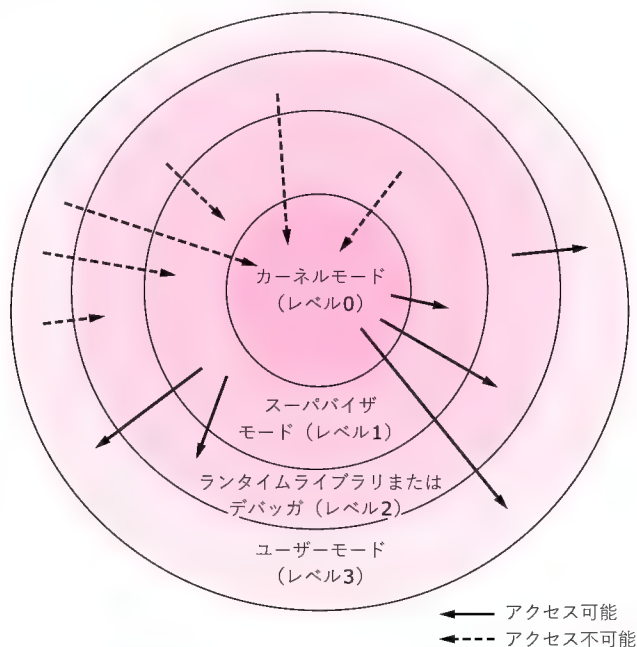
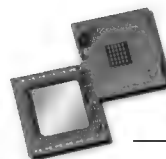
実行レベルの種類は，アーキテクチャによって異なるが，2～4種が定義されている．2～3種の場合，実行レベルに名称がついていることが多い．4種の場合は，単に，レベル0，レベル1，レベル2，レベル3と呼ぶ．値が小さいほど特権性が高い．たとえば，実行レベルの名称は次のようになっている．

2レベル：カーネル ＞ ユーザー
　　　　　スーパバイザ ＞ ユーザー
3レベル：カーネル ＞ スーパバイザ ＞ ユーザー
4レベル：レベル0＝カーネル ＞ レベル1 ＞ レベル2 ＞
　　　　　レベル3＝ユーザー

ここで，不等号は特権性の高さを表すものとする．カーネルとはOSの実行レベルであり，スーパバイザとはデバイスドライバやランタイムライブラリの実行レベルである．ユーザーとはアプリケーションプログラムの実行レベルである．多くのOSでは，カーネル（あるいはスーパバイザ）とユーザーの2レベルしか使用しない．その中間の実行レベルは，あれば便利だがOSの構造が複雑になるので，あまり使用されない．

通常，実行レベルはMPUのステータスレジスタに格納されている．一方，保護レベルはPTEで指定され，同等の情報がTLBにも格納されている．そして，アドレス変換時に現在の実行レベルとアクセスするページの保護レベルが比較され，自分と特権性が同じか，特権性が低いページであるとアクセスが許可される．アクセスが禁止されている場合はメモリ保護例外やアドレス例外が発生する．

PTEでは，リード，ライトといった，アクセスの種類での保護も可能になっている．つまり，リード可能，ライト可能，リード/ライト可能といったページ保護を独立に指定できる．アーキテクチャによっては「実行」というアクセスの種類をもっているものもある．

以上は，たいていのMPUの保護機構であるが，もっともシェアの高い（と思われる）x86アーキテクチャでは，少し事情が異なる．ステータスレジスタ（x86でいうところのフラグレジスタ）内に実行レベルを保持しない．現在実行中の仮想アドレス（セグメント）の保護レベルが，そのまま現在の実行レベルとなる．属するセグメントが変わるとき（FAR CALLや例外など）

に，移行先の仮想アドレスの保護レベルと現在の実行レベルの比較を行ってメモリ保護を実現する．

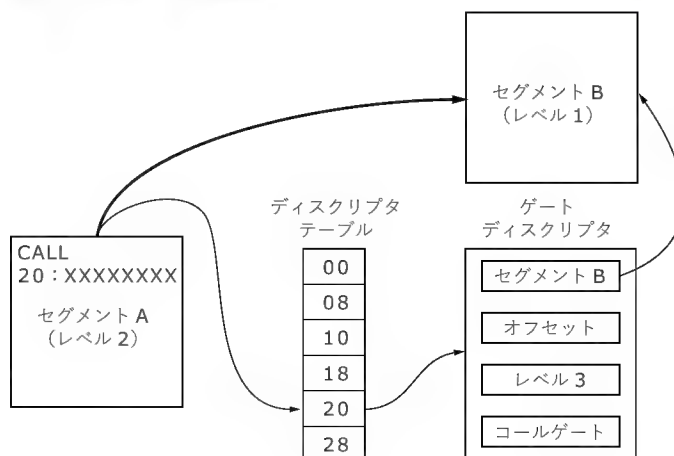
● 実行レベルの変更

MPUは，リセット直後は最高の特権性をもっている．そしてアプリケーションプログラムを実行する直前に最低の特権性に移行する．また，アプリケーションプログラムの実行中に割り込みや例外が発生すると，最高の特権性に戻る．

カーネルモードからユーザーモードへの移行は，具体的には割り込みからの復帰命令を利用する．この命令(たとえば，ERETとする)はスタックから新しいステータスレジスタの値と新しいPC(プログラムカウンタ)の値をリードして，そのPCの示すアドレスに分岐する．このとき，スタックに積んであったステータスレジスタに設定される値の中に新しい実行レベルが含まれている．アーキテクチャによってはスタックではなく，特殊レジスタからステータスレジスタとPCの値をリードするが，実質は同じである．

ユーザーモードからより特権性の高い実行レベルに移行するには，専用命令が用意されている場合もあるが，通常はソフトウェア割り込み(トラップ命令やシステムコール命令)で，一律，特権性が最高のカーネルモードに戻ることが多い．専用命令が用意されていない場合，ユーザーモードから特権性が中間のレベルに移行するのは難しく，一度，カーネルモードに移る必要がある．

ただし，x86では，また事情が異なる．仮想アドレスでの保護レベルで許可されていれば，コールゲートを使用して任意の実行レベルに移行できる．x86のプロテクトモードにおいて，セグメントレジスタの値はディスクリプタテーブルと呼ばれる，新しいセグメントとオフセットが組になったディスクリプタの集まりへの選択情報となる．コールゲートを呼び出すにはセグメント間コール(FAR CALL)やセグメント間ジャンプ(FAR JUMP)を利用する．新しいセグメントの値で選択されたディスクリプタが

〔図9〕コールゲートの概念

ゲートディスクリプタである場合がコールゲートとなる．

ゲートディスクリプタには，新しいセグメントとオフセットの値のほかに，そのゲートをコールできる(最低の)実行レベルが格納されている．FAR CALL/FAR JUMPを実行するプログラムが存在しているアドレスの実行レベルがゲートディスクリプタの実行レベルより特権性が高ければ，どの実行レベル(のセグメント)にも移行できる(**図9**)．なお，割り込みや例外が発生した場合は，(一般)ディスクリプタテーブルの代わりに例外ディスクリプタテーブルが参照され，最高の特権レベル(レベル0)に移行する．

● 仮想アドレスによるメモリ保護

ところで，メモリ保護はTLBで行うのが普通であるが，仮想アドレスの値そのもので保護を行う場合もある．すなわち，現在の実行レベルに応じてアクセスできる仮想アドレスが最初から規定されている．たとえば，MIPSのアーキテクチャがそうなっている(**図10**)．ユーザーモードでは仮想アドレスの0x80000000～

〔図10〕MIPSのアドレス空間(32ビットモード)

〔図11〕Jade(4Kp)のBAT

〔図12〕MC68030の変換制御レジスタ(TC)

31	28-27	27-25	25	24	23-20	19-16
E			SRE	FCL	PS	IS

15-12	11-8	7-4	3-0
TIA	TIB	TIC	TID

E：変換許可
SRE：SRP(スーパバイザルートポインタの許可)
　SRE＝0：：変換はすべてCRP(CPUルートポインタ)から始まる
　SRE＝1：：ユーザーアクセスはCRP．スーパバイザアクセスはSRPを使用.
FCL：FC(ファンクションコード)のルックアップ．つまり最初のディスクリプタのアクセスをFCの値でインデックスするか
PS：ページサイズ．1ページのビット数を指定
IS：イニシャルシフト．仮想アドレスのサイズ(32～17ビット)を指定．つまり，アドレス変換時に仮想アドレスの上位をマスクするビット数．32ビットなら0を指定
TISx：テーブルインデックス．TIAが1レベル目，TIBが2レベル目，TICが3レベル目，TIDが4レベル目．それぞれ仮想アドレスの中で何ビットを占めるかを指定する
注意：IS＋TIA＋TIV＋TIC＋TID＋PS＝32(ビット)の関係を保たなければならない

0xFFFFFFFFの範囲はアクセスできない．スーパバイザモードでは仮想アドレスの0x80000000～0xBFFFFFFF，0xE0000000～0xFFFFFFFFの範囲がアクセスできない．カーネルモードではすべてのアドレス空間をアクセスできる．

そもそも組み込み制御分野では，アドレス変換が必要ない場合が多い．メモリ保護さえあればよい．このような要求に対応するためにMIPSのJade(4Kp)では，BAT(Block Address Transfer)と呼ばれる機構を提供している(**図11**)．これは，仮想アドレスは基本的に物理アドレスと同じになり，メモリ保護だけは**図10**と同等になる機構である．あるいは，MC680x0では現在の実行レベルやアクセスの種類がファンクションコード(FC2，FC1，FC0)としてMPUの外部端子に出力されている．この信号とアドレスバスの値を外部回路で処理して，保護違反のアクセスを検出すると，バスエラーをMPUに通知できるしくみを提供している．

6 MMUの実例

ここでは，いくつかのMPUでMMUの実例を見てみよう．

● MC68030/MC68040のMMU

▶アドレス変換

MC68030ではアドレス変換は0～5レベルの範囲で自由に設定できる．その設定を行うためのレジスタが変換制御レジスタ(TC)である．**図12**にTCの形式を示す．TIA，TIB，TIC，TIDでそれぞれ1レベル，FC(Function Code＝保護レベル)ルックアップを行えばさらに1レベル増えるので，最大5レベルのページングとなる(インダイレクト指定をすれば6レベルまで可能ということであるが，ここでは触れない)．TIA，TIB，TIC，TIDの値を0に設定することで1～4レベルのページングが可能になる．ゼロレベルというのは，ルートポインタ(CRP，SRP)の中に直接変換後の物理アドレスが指定されている場合(アーリーターミネーションという)である．

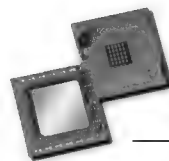
MC68030では，TLB(ATC)ミスに際し，

CRP(SRP)→(FC)→TIA→TIB→TIC→TID→

と，変換テーブルをたどっていき(テーブルサーチ)，最終的にページテーブルに到達する．5レベルのアドレス変換例を**図13**に示す．この例ではPSは256バイト(8ビット)で，TIA，TIB，TIC，TIDはそれぞれ4ビット(各テーブルは16エントリ)である．**図14**にはFCルックアップを用いたテーブルサーチ例を示す．FCによって，次にアクセスするレベルAテーブル(TIAでインデックスされるテーブル)のベースアドレスを個別に設定できるので，メモリ保護が実現できる．

MC68030のMMUでは変換テーブルをサーチするために，仮想アドレスを細かく分割しすぎている感もある．実際的には2～3レベルのページングしか行われないので，オーバスペックとも思える．後継のMC68040ではこの点が改良された(退化と呼ぶ人もいるが)．テーブルサーチは3レベルに固定し，ページサイズも8Kバイトまたは4Kバイトのみが許されている．テーブルインデックスはレベルAテーブル，レベルBテーブルが7ビット，レベルCテーブルが5ビット(1ページ8Kバイトの場合)，または，6ビット(1ページ4Kバイトの場合)である．MC68030風にいえば，

IS ＝0
TIA＝7
TIB＝7
TIC＝5(8Kバイト/ページ)，6(4Kバイト/ページ)
PS ＝13(8Kバイト/ページ)，12(4Kバイト/ページ)

ということになる．

〔図13〕MC68030の5レベルのテーブルサーチ

▶ATC(TLB)

MC68030のATCは，22エントリのフルアソシアティブ構成である．仮想アドレスとFCを組で検索し物理アドレスを得る(**図15**．ただし，この図は機能から推測した予想図)．一方，MC68040のATCは64エントリの4ウェイセットアソシアティブ構成である．仮想アドレスとFCの最上位ビット(FC[2])を組で検索し物理アドレスを得る(**図16**)．FCの最上位しか見ないということは，スーパバイザとユーザーを区別するだけで，命令とデータの区別を行わないことを意味する．

● i486のMMU

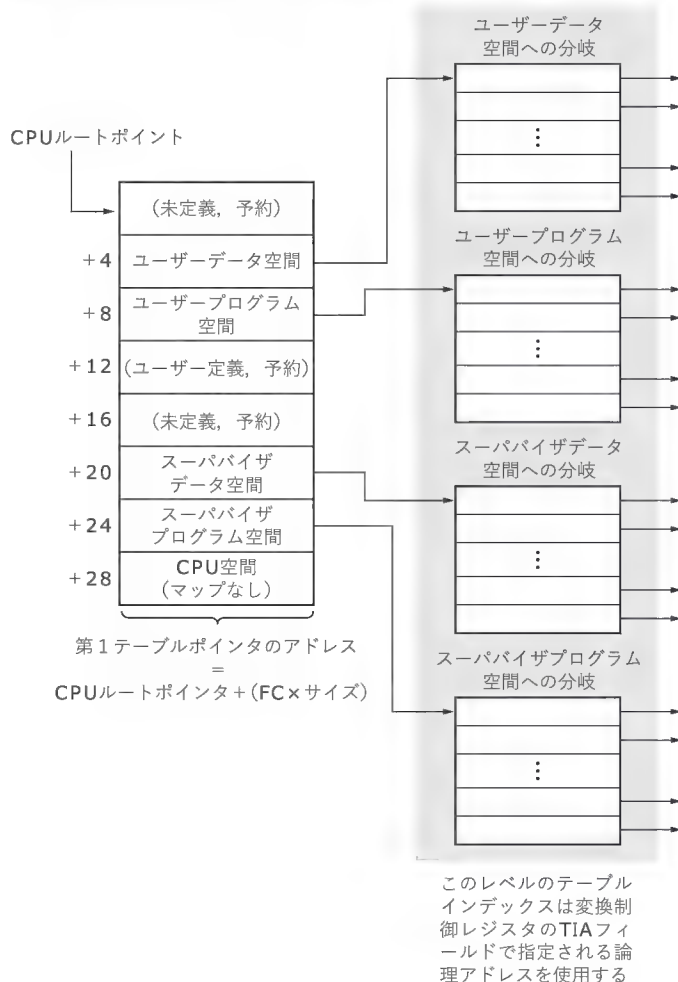
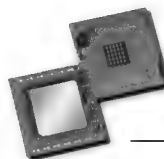
Pentium以降，IntelのMPUの内部構造に関して詳しく記してある資料は少ない．Intelアーキテクチャはi386で完成しているので，MMUの基本構造もそれ以降大きな変化はないと予想される．ここでは，i486のMMUを**図17**に示す．

TLBは全32エントリの4ウェイセットアソシアティブ構成を採る．セグメントユニットによって生成されたリニアアドレスのビット14〜12でTLBのエントリ(セット)をインデックスし，そこから選択される四つのタグブロックの値と，リニアアドレスのビット31〜15を比較する．もし，どれかと一致すればヒットであり，どれとも一致しなければミスである．ヒットする場合は，そのウェイとセットに対応するデータを物理アドレスのページアドレスとしてアドレス変換を行う．ミスの場合は，MPUはメモリ上のアドレス変換テーブルを検索し，リニアアドレスに対応する変換情報を，TLBの指定されたエントリに格納する．そして，再びアドレス変換を試みる(当然，次は必ずヒットする)．このとき書き潰されるウェイは，疑似LRUにより，もっとも参照された頻度が少ないものが選択される．

〔図14〕MC68030のFCルックアップをもちいたテーブルサーチ

〔図15〕MC68030のATC（筆者想像図）

〔図16〕MC68040のATC

〔図17〕i486のTLB

● MIPSアーキテクチャのMMU

▶TLBの概要

MIPSアーキテクチャのMMUにはTLBしかない．MPUは仮想アドレスがTLB内にあるか否かを検索し，ヒット(ある)すれば対応する物理アドレスを供給する．ミス(ない)あるいは保護違反を検出する場合はTLB例外を発生するのみである．

TLBミスが発生してもアドレス変換テーブルを自動的に検索し，TLBのエントリを入れ替えるという操作は行わない．代わりにTLBの内容をソフトウェアで操作できるようになっており，TLBミス発生時のエントリ入れ替え処理はソフトウェアで行うことになっている．エントリ入れ替えのための複雑なハードウェアは省略するというRISCならではの考え方である．MIPSアーキテクチャの発表当時，TLBの更新は数命令で実現可能であり，システム性能の低下はないと明言されていた．

R4000以降，MIPS系のMPUは64ビットプロセッサであり，アドレス空間に関して32ビットモードと64ビットモードをもっている．TLBの各エントリも32ビットモードと64ビットモードで若干異なる．**図18**にTLBエントリの形式を示す．各エントリは特権レジスタである，エントリHi，エントリLo0，エントリLo1，ページマスクレジスタと直接対応する領域をもっている．

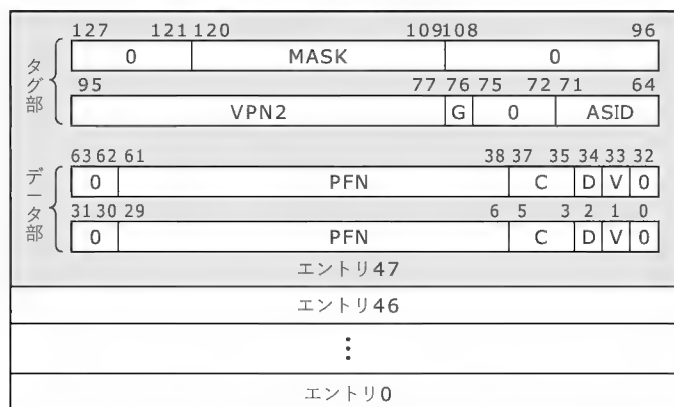
TLBは，32ビットモードにおいては32ビットの仮想アドレスを，64ビットモードにおいては64ビットの仮想アドレス(TLBには40ビット分の領域しかないが)を，通常は36ビットの物理アドレスに変換する．このしくみを**図19**に示す．物理アドレスのビット数はMPUによって異なり，それによってエントリLo0，エントリLo1レジスタ内のPFN領域のビット数が決定されている．

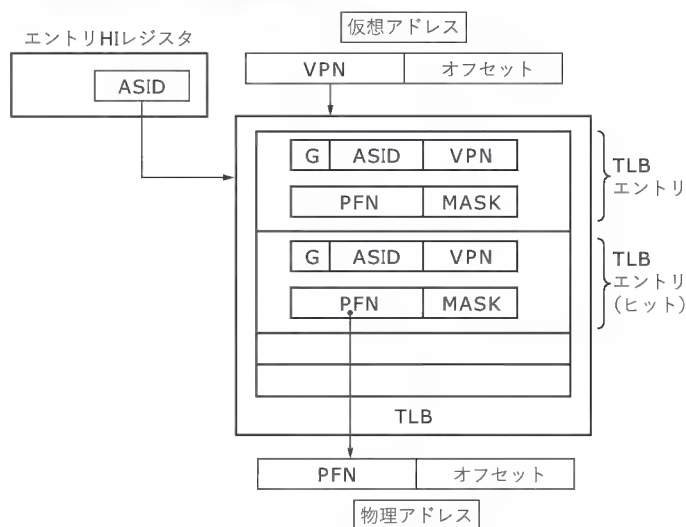
MMUのサポートするページサイズは，エントリごとに4Kバイトから16Mバイトの範囲で4の倍数ごとに指定できる．これは，エントリへの書き込み時にページマスクレジスタで指定する．アドレス変換時に，仮想アドレス番号(VPN)の下位ビットをページマスクレジスタの値で無視して仮想アドレスの検索(一致比較)が行なわれる．

▶ダブルエントリ構成

MIPS系のMMUの大きな特徴は，二つのページを組にして扱う点である．TLBは48エントリ(R4200，R4300では32エントリ)のフルアソシアティブ構成で，1エントリは連続する2ページ分(偶数ページと奇数ページ)を示す一つの仮想アドレスと，それに対応する二つの物理アドレスを保持している．

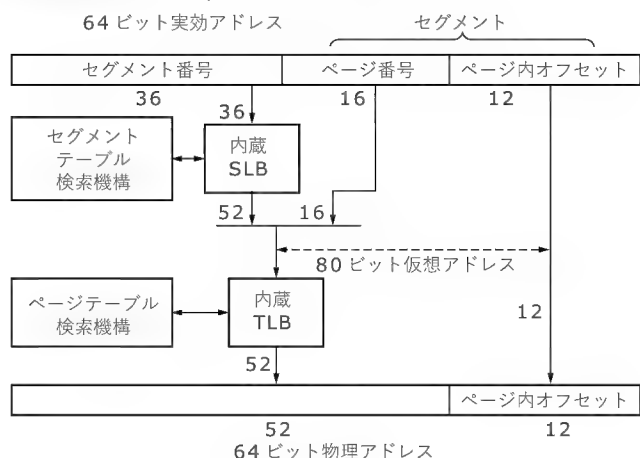
仮想アドレスはエントリHiレジスタ，偶数ページ/奇数ページに対応する物理アドレスは，それぞれ，エントリLo0レジスタ/エントリLo1レジスタで指定する．このTLB形態は一般にダブルエントリ形式と呼ばれている．48エントリではあるが，実質的には，98エントリ相当，あるいは指定したページサイズの2倍のページサイズをもっているとみなせるためTLBのヒット率が高くなるといわれている．

〔図18〕MIPSのTLB(R4000/32ビットモード)

MASK：ページ比較マスク，VPN2：仮想ページ番号，G：グローバル(0のとき，TLBルックアップ時にASIDを比較)，ASID：アドレス空間ID：PFN：ページフレーム番号(物理ページの上位ビット)，C：キャッシュアルゴリズム，D：ダーティ，V：バリッド

〔図19〕MIPSのアドレス変換

- エントリHIレジスタには現在のアドレス空間ID(タスクID)を格納しておく
- 仮想アドレスのVPN(ページ番号)とASIDがTLBの各エントリのVPN，ASIDと比較される．ASIDを比較するのはGビットが0の場合のみ．比較時にMASKビットでVPNの下位ビットをマスクすることで種々のページサイズに対応する
- 比較内容が一致すると，PFN(物理アドレスの上位ビット)を取り出す
- オフセット部分がTLBを通さずに使用され，PFNと結合して物理アドレスを生成する

▶ASID

なお，エントリHiレジスタは仮想アドレスの他にタスク番号に対応するASID(Address Space ID)を指定できる．TLBの各エントリにもASIDに対応する領域があり，仮想アドレスの検索時にASIDの一致も調べられる．仮想アドレスとASIDが一致して初めてヒットとなる．このため，マルチタスク環境下においてタスク切り替えが生じても，エントリHiレジスタのASIDを変更してさえおけば，TLBエントリの無効化をする必

〔図20〕PowerPC(64ビットモード)のMMU

要がない．

たとえば，あるタスクの仮想アドレス0番地と，他のタスクの仮想アドレス0番地に対応する物理アドレスは一般には異なるので，ASIDがなければ，0番地を他のタスクの物理アドレスに変換してしまう必要がある．このため，ASIDをもたないTLB構成においては，タスク切り替え時にTLBの全エントリを無効化しておく必要がある．OSやライブラリ空間など各タスク間で共有する領域に関しては，TLBエントリのグローバルビットをセットしておけばよい．この場合，仮想アドレスの検索はASIDを無視して行う．

また，エントリLo0，エントリLo1レジスタは，対応するページのキャッシュ情報，保護情報も指定できるようになっている．

▶入れ替え方式

MIPSアーキテクチャでは，TLBミスが生じた場合，エントリの入れ替えはソフトウェアで行う．どのエントリを追い出すかはランダム(任意)に決定する．といっても実際には，ランダムレジスタという特権レジスタが指し示すエントリを更新することになる．ランダムレジスタは，ワイヤードレジスタ(特権レジスタ)で示される値と(TLBエントリ数−1)の間の任意の値を保持している．つまり，ランダムレジスタは0から(ワイヤードレジスター1)の値は指し示すことがないので，TLBのエントリ0から(ワイヤードレジスター1)までは決して追い出されることのない安全な(必ずヒットする)エントリとして確保できる．

● PowerPC(64ビットモード)のMMU

2002年秋のMicroprocessor Forumで，IBMからPowerPCの64ビット実装であるPowerPC 970が発表された．現在ではPowerMac G5に搭載されているMPUである．

64ビットPowerPCといえば，かつてPowerPC 620が計画されたが実現には至らなかった．ただし，マイクロアーキテクチャを少し変更してPower3として登場した．PowerPCの64ビット実装の最初は，1995年に登場したA30と呼ばれるAS/400用のMPUである．A30は1個のCMOSチップと6個のBiCMOSチップの合計7チップからなるPowerPC唯一のマルチチップ実装であえる．A30は1997年にシングルチップ実装のPowerPC RS64に置き換えられた．その後RS64-II，RS64-III，RS64-IVと改良が続けられ，AS/400，RS/6000 S80シリーズなどのビジネス用サーバとして利用されている．A30はサーバ用なので，本来のPowerPCとは言えない．

さて，PowerPCの64ビット実装では，マシン状態レジスタのビット指定により，64/32ビットモードを切り替えることができる．セグメントサイズとページサイズは32ビットモードと同様で，それぞれ，256Mバイトと4Kバイトである．このため，64ビットモードではセグメント数(16個から64G個)とセグメントのビット幅の拡張(24ビットから52ビット)により仮想アドレス空間を実現する．

なお，PowerPCではユーザーが使用する64ビットの仮想アドレスを実効アドレスと呼び，システムが管理する80ビットの仮想アドレスと区別している．アドレス変換機構により，80ビットの仮想アドレスが64ビットの物理アドレスに変換される．

PowerPCでは実効アドレスの上位4ビットでセグメントレジスタを選択していたが，64ビットモードでは実効アドレスの上位36ビットで選択する．36ビットといえば64G個と膨大な数からの選択となるため，すべてを主記憶にもっていたのでは主記憶があふれてしまう．そこで，主記憶上にページテーブルと同様な形式のセグメントテーブルを置き，MPU内部にTLBと同様のSLB(Segment Look-aside Buffer)を内蔵してアドレス変換を行う．アドレス変換時に実効アドレスがSLBに存在しない場合，TLBミスの発生時と同様にセグメントテーブルが自動的に検索されてSLBのエントリを置き換える．

図20に64ビット実装時のPowerPCのアドレス変換を示す．

まとめ

MMUというものが，だいたいどのような働きをするものか理解していただいただろうか．以上をおさえておけば，基礎知識としては十分である．個人的にはx86アーキテクチャに思い入れはないが，図らずもx86のアーキテクチャの説明がかなりの部分を占めてしまった．アドレス変換やメモリ保護に付いてはx86のやり方は特異にみえるが，現在のコンピュータアーキテクチャを語る上ではこれも必須な教養であろう．

なかもり・あきら フリーライタ

Appendix 1

クロック周波数の上限は何で決まるのか

高速化技術の基礎

中森　章

● 高速化とはどういうことか？

「MPUが高速」という場合は，一般には処理性能が高いことを示す．MPUの処理時間は，次の式で表される．

処理時間＝（処理に要する総クロック数）/（クロック周波数）
＝（処理に要する命令数×CPI）/（クロック周波数）

処理性能が高いということは，この処理時間が短いということである．処理に要する命令数を減らすことで処理時間を短くするのがCISCのアプローチであった．反面，CPI（Clocks per Instruction）を減らすこととクロック周波数を高くすることがRISCのアプローチであった．最近のMPUはRISC化しているので，RISCのアプローチをメインに考える．

CPIを減らすこと，つまり，IPC（Instructions per Clock）を増やす工夫は，先月号の特集で説明したパイプライン，スーパースカラなどで行われている．クロック周波数を上げる工夫は，一部はスーパーパイプラインで行われている．しかし，これらマイクロアーキテクチャ的なアプローチはすでに出尽くした感がある．そこで本稿では，クロック周波数を上げるという観点で考えてみたい．以下，「高速」という言葉は**クロック周波数が高い**という意味で使用する．

● クロック周波数を規定する要因

クロック周波数とは，クロックが単位時間に変化する回数を示す．MPUはクロックに同期して動作しているため，クロック周波数が処理速度を規定する．つまり，クロック周波数を高くすればするほど，MPUは高性能になる．しかし，クロック周波数は無条件に高くはできない．それは，MPUの内部回路を電気信号が伝わる時間（電流の速度）に依存するからである．つまり，クロックが1回変化する時間（これが1周期＝周波数の逆数）に電気信号が移動できる距離が，クロック周波数の物理的な限界である．しかも内部回路には，次のように電気信号の流れを妨げるいろいろな要因が存在する．

▶クリティカルパス

クリティカルパス（critical path）とは，二つのフリップフロップ（クロックが供給されるラッチ）間の配線で生じる最大遅延時間のことである．MPUの内部回路はクロックに同期して動くため，クロックの1周期の間に，あるフリップフロップから別のフリップフロップに電気信号が伝わらないと誤動作する．現実には，フリップフロップ間には，何段階にもわたってAND/OR/XOR/NOTといった論理ゲートが存在する（**図A**）．電気信号はこれらの論理ゲートを通過するたびに少しずつ遅延（ゲート遅延）が生じる．また，配線自身の抵抗によっても遅延（配線遅延）が生じる．これらを合計した遅延時間がクロックの1周期の時間より小さくないと，誤動作する．

高速化を実現するための基本は，ゲート遅延を低減するために，フリップフロップ間の論理ゲートの段数を減らすことである．これは論理設計の役割である．あるいは，配線遅延を低減するために，配線を短くすることである．これは回路設計の役割である．とはいえ，一つの配線が多くの論理ゲートを通過していると，論理ゲート用にある程度の距離が必要なので，配線を短くするためには論理ゲートの段数を減らすことも必要である．

▶クロックスキュー

クリティカルパスはフリップフロップ間の電気信号の遅延によって規定されるが，これは，すべてのフリップフロップに対して，クロックが同じタイミングで変化することを前提としている．しかし，クロックも配線によって伝達されるので，配線の形状や長さによってバラツキを生じる．このクロックのバラツキをなくすために，クロックの配線にバッファを入れたり遅延素子を入れたりして遅延をそろえることが行われる（**図B**）．

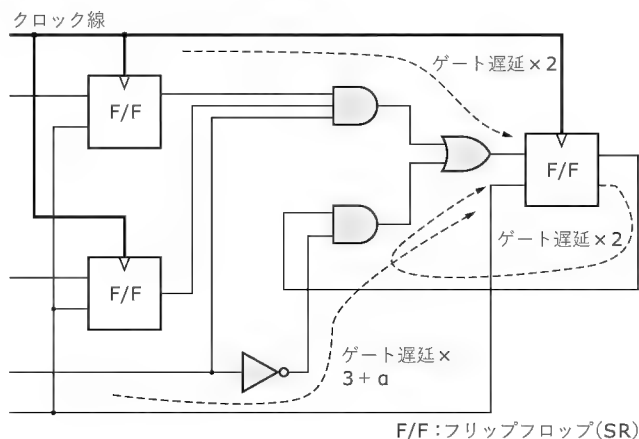
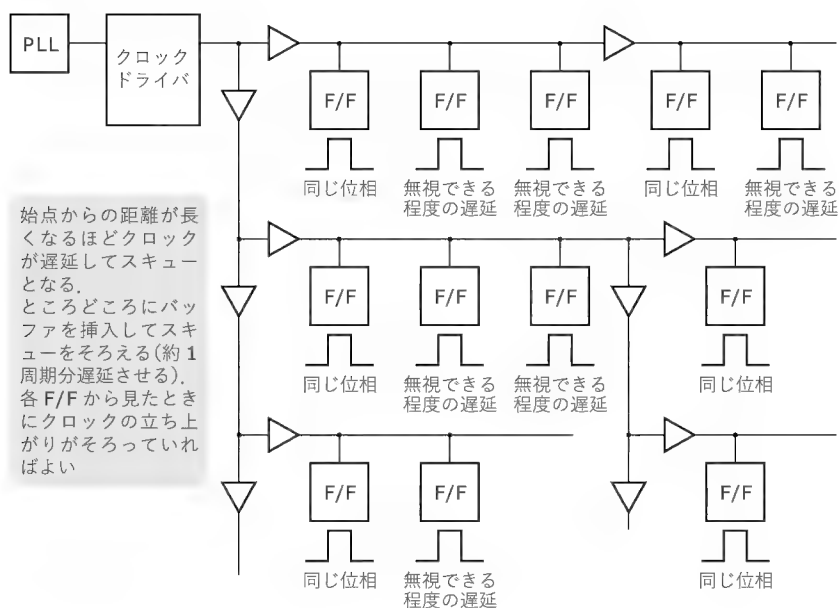
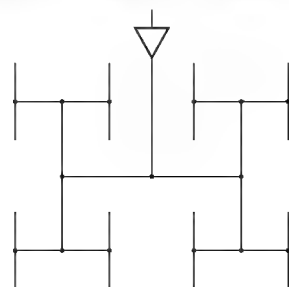
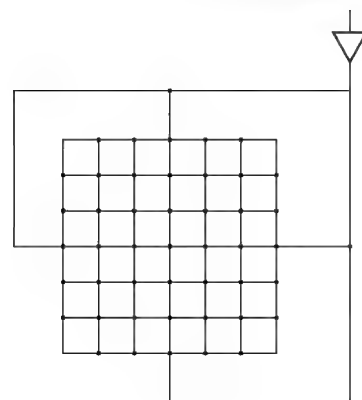
しかし，クロックのバラツキを完全に一致させることは不可能なので，一致させられなかった分が**クロックスキュー**となる．クロックスキューは，フリップフロップを伝わる電気信号から見ると遅延とみなされるので，クロック周波数低下の要因となる．

クロックスキューを揃えるために一般的に使用されるのは，「H-Tree」と呼ばれる手法である．これはクロックラインをアルファベットのH形状に配線することで，クロックドライバからクロックの供給先までの距離をそろえる（**図C**）．インテルのPrescottではH-Tree構造を改良することで，Northwoodでは22psであったクロックスキューをPrescottでは7psに低減している．

ARM10の製造で1.2GHzを達成したSumsung社は，H-Tree構造ではなく，「メッシュ」構造を採用している．これはクロックをメッシュ状に配線し，その各辺から同時にクロックを供給する手法である（**図D**）．H-Treeに比べるとクロックスキューは小さくなるが，同時に駆動するラインが多いため，消費電力が大きくなる．

▶ゲート遅延

ゲート遅延はトランジスタ遅延ともいわれる．つまり，トランジスタのソースからドレインに電気信号（NMOSトランジスタの場合は

〔**図A**〕フリップフロップと論理ゲート

〔図B〕クロックスキュー

〔図C〕H-Tree型のクロック配線

〔図D〕メッシュ型のクロック配線

電子，PMOSトランジスタの場合は正孔)が流れる場合の遅延時間のことである．これはゲートの下に形成されるチャネル長に比例して大きくなる．したがって，ゲート長(やチャネル長)を短くできる微細プロセスを使用すればゲート遅延を低減できる．

ゲート遅延はトランジスタの製造条件，すなわち，V_{th}(Threshold Voltage，V_tとも呼ぶ)，I_{on}(high drain current)，I_{off}(low off state leakage)などにも依存する．V_{th}とはソースからドレインに電流が流れ始める電圧のことで，この電圧が低いほどトランジスタは高速にON/OFF動作する．I_{on}とは，トランジスタがON時の単位長あたりの電流値(通常，μA/μmという単位で示される)のことで，この値が大きいほど遅延が少なくなる．

I_{on}を大きく製造するということは，トランジスタの電流駆動能力を大きく製造することである．I_{off}とは，トランジスタがOFF時のソースからドレインに流れるリーク電流のことで，消費電力を下げるためにはI_{off}の低減が必須である(I_{off}の単位は，通常nA/μmなので，I_{on}に比べると非常に小さいのだが)．リーク電流はトランジスタのON電圧を邪魔するので，I_{off}の低減はV_{th}の低下を可能にする．リーク電流を抑えるためにSOI(Silicon On Insulator)という技術などが提案されている．

以上のことから，微細プロセスでゲート長を短くし，V_{th}が低く，I_{on}が大きいトランジスタを製造すれば，高速なON/OFF動作が可能になる．近年，THzトランジスタ(TeraHertz Transistor)，すなわち1THz以上のクロック周波数で動作するトランジスタの発表が花盛りであるが，これはゲート遅延が1ps以下であることを示しているにすぎない．つまり，ゲート遅延の逆数をクロック周波数とみなしているのである．本来のクロック周波数は回路の配線遅延などにも依存するので，それがそのままMPUの動作周波数にはならない．

ところで，温度が上昇するとチャネルの熱抵抗が増加するのでゲート遅延が増加する．そこでトランジスタを冷却することでクロック周波数を高めることも考えられる．半導体の製造メーカーは，ある決められた温度範囲内で動作クロック周波数を保証しようとするため，無理矢理冷却することでクロック周波数を向上させることは想定していない．しかし，PCの自作などでのクロックアップでは，メーカーの保証外ではあるが，とにかく冷やすのは常套手段である．

▶歪みシリコン(Strained Silicon)

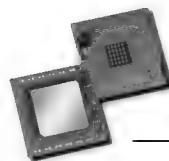
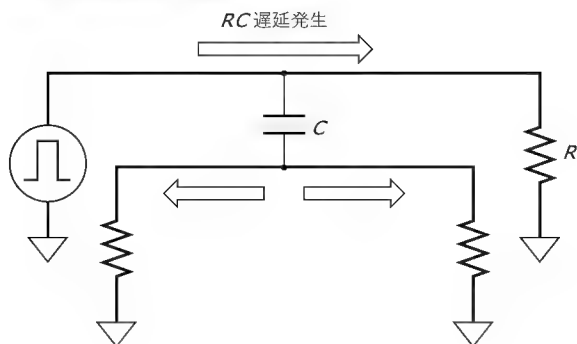
MOSトランジスタの微細化にともない，ゲート酸化膜が薄くなるとリーク電流が増加するため，それを抑えるためのhigh-k(誘電率の高い)絶縁膜の研究がさかんである．しかし，high-k絶縁膜では，SiO_2に比べ，電気信号(電子や正孔)の移動速度が低下してしまう．そこで，電気信号の移動度を高める手段として**歪みシリコン**が注目されている．

シリコン単体より格子定数の大きいSiGe結晶の上に薄いSi膜を作ると格子が引き延ばされて歪むが，この状態では通常のシリコンに較べて電子，正孔の移動度が向上することが知られている．しかし，無理な力を加えて製造するので格子欠陥ができやすいという問題がある．また，歪みシリコンによるキャリア移動度の向上度合いが電子と正孔で差があることや，NMOSトランジスタではしきい値が低下するため，回路設計が難しいという問題もある．そのためか，歪みシリコンを利用したMOSデバイスが発表されたのは2001年と意外に新しい．本格的な実用化はまだ先の話である．

……と思っていたところ，インテルはゲート長50nm(ゲート酸化膜厚1.2nm，設計ルールは90nm)のCMOSプロセス「P1262」で歪みシリコンを採用することを明らかにした(2002年8月13日)．インテルの試算ではトランジスタの電流速度が10～20%向上する．このP1262はPentium4系のPrescottなどに適用されている．

歪みシリコンといえば，IBMがいちばん先行していると考えられている．そのIBMですら65nm世代から適用と表明していたが，後

〔図E〕配線の寄生容量

〔図F〕ゲート遅延と配線遅延

発のインテルが，1世代早い90nmから適用すると発表して注目を集めている．なお，IBMによると歪みシリコンを用いた場合の動作速度は最大35％向上するという．

▶配線遅延

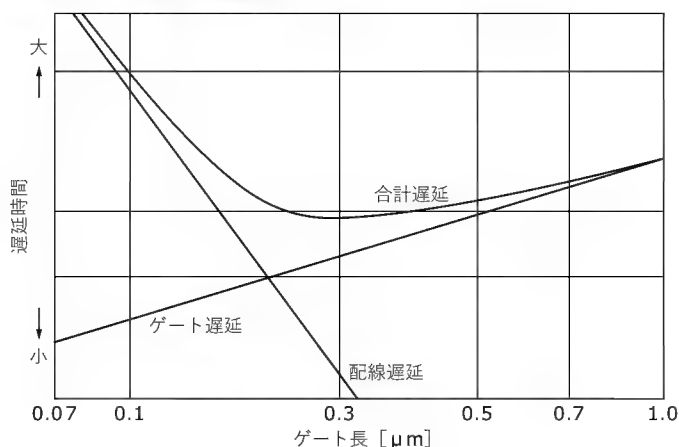
遅延はトランジスタだけでなく，配線自身によっても発生する．配線を伝わる電気信号の遅延は，配線抵抗（*R*：resister）と配線容量（*C*：capacitance）の積で決定される（*RC*遅延）．電気信号はデバイス中を伝達する際に，配線にぶら下がっている容量（キャパシタンス＝コンデンサ）を充放電させながら伝搬される（**図E**）．

配線抵抗を下げるためには，配線幅を広くし，配線の厚さを厚くしなければならない．しかしそれでは微細化に反する．通常のMPUではアルミニウムで配線を行うが，最近は，より電気抵抗の小さい，銅を用いて配線するのが流行である．銅を使用すれば，細い配線にしても電気信号がスムーズに流れる．しかし，銅配線を使用するためにはダマシン（damascene）プロセスという新しい配線形成方式が必要になるので製造の手間がかかる．このため，現在では真に高速を要する場面でしか使用されない．

また，製造プロセスの微細化により，配線の寄生容量が増加する傾向にある．配線間の寄生容量を減らすためには，配線間隔を広くする必要がある．これらはプロセスの微細化とは逆行する傾向である．その解決策が多層配線である．これは，高集積化された配線層を，層間膜を挟んで，何層にも積み重ねて配線系を実現するものである．

現在，層間膜としは二酸化シリコン（SiO_2）が主流である．しかし，寄生容量を低減するために，さらに誘電率の低い（low-k）材質であるHSQ膜，アモルファスカーボン膜などが注目されている．SiO_2の誘電率は4.1であり，一般にlow-k材質といえば，誘電率が3.0以下の材質を意味する．

配線材料の低抵抗化は，銅配線の採用でほぼ物理限界に来ている．このため，層間絶縁膜材料の低誘電率化は高速配線実現の要として，ますます重要になっている．

従来，回路の遅延はゲート遅延が大勢を占めていたが，製造プロセスが微細化するのにともない，配線遅延が無視できなくなってきた（**図F**）．製造プロセスの微細化は，ゲート遅延を低下させるが，反対に配線遅延を増加させる．その境目はゲート長が0.18μmあたりにあるといわれている．

▶動作電圧

最近，製造プロセスの微細化にともない，**動作電圧**も低下する傾向にある．基本的には，CMOS回路は面積が小さいほど低い電力で動作できる．また，CMOS回路は電圧が高いほどスイッチング速度が速い．あるいは，CMOS回路の動作速度は，NMOS，PMOSトランジスタの電流駆動能力に依存する．

電流駆動能力は，大雑把にいうと，ゲートにかかる電圧とV_{th}の差に比例する．このため，動作電圧を高くすれば高速動作が可能である．しかし，あまり大きな電圧を加えるとトランジスタが破壊されてしまう．製造プロセスが微細化するにしたがって，この限界電圧（絶対最大定格という）も低下する傾向にある．また，微細化により寄生容量の影響が支配的になり，電圧の増加に対する高速動作への寄与も飽和状態にある．したがって，むやみに高い電圧を加えればいいというものでもない．

ところで，動作電圧を上げるということは消費電力が増大することでもある．むしろ，動作電圧の決定には，動作速度よりも消費電力の限界のほうが支配的になっている．したがって，近年の研究では，動作速度をある程度維持しつつ，いかに消費電力を下げるかが課題になっている．つまり，基本的には，動作電圧を下げるとともに，V_{th}も下げるという手法が採られる．V_{th}を下げると，リーク電流が増大する傾向にあるので，何らかの対策が必要なのは上述のとおりである．

▶*IR*ドロップ

製造プロセスの微細化が配線の抵抗を増大させることはすでに述べた．このため，電源からトランジスタまでの距離が長くなると，配線抵抗によって電圧降下（*IR*ドロップ）が発生する．これは，トランジスタの動作電圧を下げることに等しく，動作速度の低下につながる．このため半導体回路では電源構造の配置も重要になる．

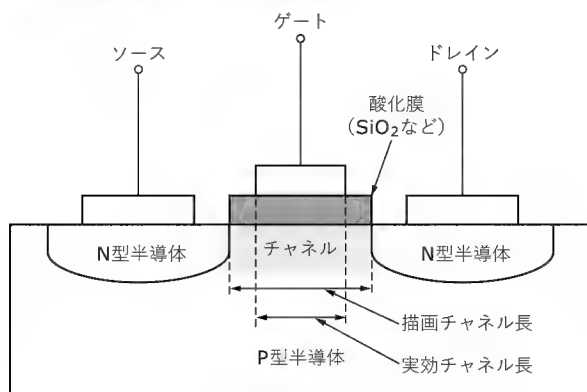
● 製造プロセスの微細化

▶プロセスルール

半導体の製造プロセスは「0.13μm」などと長さを指定して表現する．この値が小さいほど微細な製造プロセスということができる．長さの単位は，2001年くらいまではμm（マイクロメートル＝ミクロン）が主流であったが，2002年以降は，プロセスの微細化が進んだため

〔図G〕NMOSトランジスタの断面

〔表A〕インテルの製造プロセスの変遷

導入年	製造プロセス(世代)	ゲート長	動作周波数	動作電圧
1993	0.5 μm	0.50 μm	150～200MHz	3.3V
1995	0.35 μm	0.35 μm	233～300MHz	2.5V
1997	0.25 μm	0.20 μm	300～533MHz	1.8V
1999	0.18 μm	0.13 μm	500～2000MHz	1.5V
2001	0.13 μm	70nm	1.3～2.4GHz	1.3V
2003	90nm	50nm	3.0～5.0GHz	1.1V
2005	65nm	30nm	10GHz?	0.85V
2007	45nm	20nm	20GHz?	0.70V
2009	30nm	15nm	50GHz?	0.60V
2011	20nm?	10nm?	100GHz?	?

nm(ナノメートル)で表現することが多くなった．

この長さはトランジスタのゲートの幅を表すものである．ソースとドレイン間のチャネルの長さにも等しいので，**チャネル長**ともいう．そして，このチャネル長には，リソグラフィ技術(写真のようにマスクパターンをシリコンウェハに露光する技術)で物理的に形成される**描画チャネル長**と，ソースとドレインに実際に電圧を加えた場合に電気的に形成される**実効チャネル長**の2種類がある(**図G**)．現実的に，描画チャネル長のほうが実効チャネル長よりも大きい．

製造プロセスを表す場合にどちらの基準を使用するかは，国やメーカーで異なる．一般的には，米国メーカーは描画チャネル長を使用し，日本メーカーは実効チャネル長を使用する傾向がある．しかし，このような状況は混乱をまねくので，世界的には，描画チャネル長で基準を統一する動きがある．将来的には，すべて描画チャネル長に統一されると予想される．具体的には，描画チャネル長が製造プロセスの世代を表し，実効チャネル長でゲート長を表す．

ゲート長が短いほど高い動作周波数を実現できる．キャリアが移動する距離が短いほどゲート遅延が少なくなるからである．

表Aにインテル社の製造プロセスの変遷を示す．動作周波数はPentiumを基本としているが，Pentium4になってパイプラインのステージ数が増えたので，2000年以降は上限の動作周波数が一気に上昇している．

▶ムーアの法則

製造プロセスが微細化するに伴ってトランジスタの面積は小さくなる．このため，MPUの1チップに集積できるトランジスタ数は相対的に多くなる．トランジスタ数が多いということは実現できる機能が多いということであり，同時に微細化により動作周波数も上昇するので，MPUを使えば何でもできるという世界が近付きつつある．このような半導体の進歩は，インテルの創立者の一人であるGordon Moore氏が1975年に提唱した，「チップに集積可能なトランジスタ数は18～24か月ごとに倍増する」というムーアの法則にも示されている．

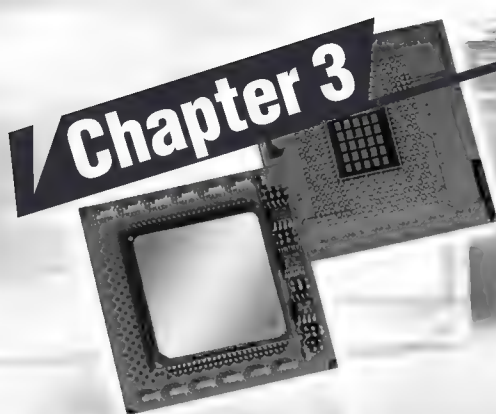
しかし近年，このムーアの法則の有効性がいつまで維持できるかが注目を集めている．「技術的にあと2～3世代(6年)が限界」，「18～24か月というペースに限界が出てきた」，「限界はまだまだ，問題は手段である」など，さまざまな見方がある．ところが，2001年になってからこの限界説が和らいできている．少なくとも後10年，つまり今後5世代に関してはムーアの法則は維持されるだろうと見るアナリストが多くなった．

ムーアの法則自体は経験則なので根拠があるわけではない．半導体がムーアの法則にしたがって進化しているというよりも，半導体メーカーがムーアの法則を維持するために，それを目標として，製品開発をしているといったほうが正確であろう．

現在の半導体は，遠紫外線を使用するリソグラフィ(lithography)技術により，透明な石英板の上にクロム(Cr)でトランジスタや配線のパターンが形成されているマスク(レチクル：reticle)の模様を，シリコンのウェハに転写して回路を生成する．この技法では，今後1～2世代に相当する100nmまでのパターンまでしか対応できないとされている．2002年時点，半導体メーカーは0.13μmプロセスが主流であり，限界に近づきつつある．リソグラフィの光源として，0.13μmまではKrFエキシマレーザが使用され，0.13μm～0.10μm(＝100nm)まではArFエキシマレーザが有力候補とされている．これに代わる新しいリソグラフィ技術がなければ，半導体メーカーは2004年か2005年に壁に突き当たり，それ以上MPUの集積化(＝高速化)ができなくなってしまう．その候補としては F_2 エキシマレーザ(波長：157nm)，等倍X線，縮小X線，電子ビームなどがある．それぞれ一長一短があり，決定打はない．

＊　　＊

ここでは主として，トランジスタレベルでの高速化技術をまとめてみた．この分野は日進月歩なので，まだまだ技術革新を遂げていくと考える．

なかもり・あきら　フリーライタ

外的要因と内的要因，ハードウェア割り込みとソフトウェア割り込みの違いを理解する

割り込みと例外の概念とその違い

中森　章

割り込みには，MPUの動作とはまったく非同期に外部のデバイスが要求するハードウェア割り込みと，プログラム中に明示的に分岐命令を記述するソフトウェア割り込みがある．また，プログラムの実行結果によって発生する予期しない事象を例外と呼ぶ．ハードウェア割り込みは外的要因で発生するが，ソフトウェア割り込みと例外はMPUの内的要因で発生する．例外と割り込みの区別はそれぞれのMPUアーキテクチャ上の決め事であり，その本質は同じと考えられる．　（編集部）

はじめに

MPUには割り込みという概念がある．15年ほど前，筆者は割り込みというものの概念がよくわからなかった．MPUは与えられた処理を順次こなしていく．その処理に割り込んでいったい何をするのか．処理Aをこなしながら処理Bも行う必要があるなら，AとBを同時に実行するようにプログラムすればよいではないか．

例外についても然り．行っていることは，とどのつまり固定アドレスに分岐して戻ってくること．それはサブルーチンコールと何が違うのか．

以降の解説は，15年を経て筆者が感じ取った割り込みと例外の意義やしくみである．

1 MPUにおける割り込みと例外

● 割り込みとは何か

割り込みとは，一連の仕事をしているときに，その仕事を中断させて別の仕事をさせることである．割り込みをされる側からは，予期しないタイミングで発生するのが特徴である．

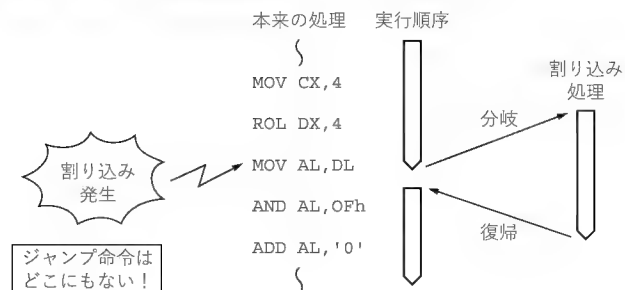
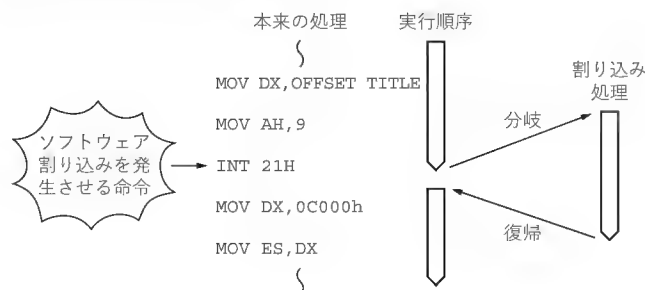
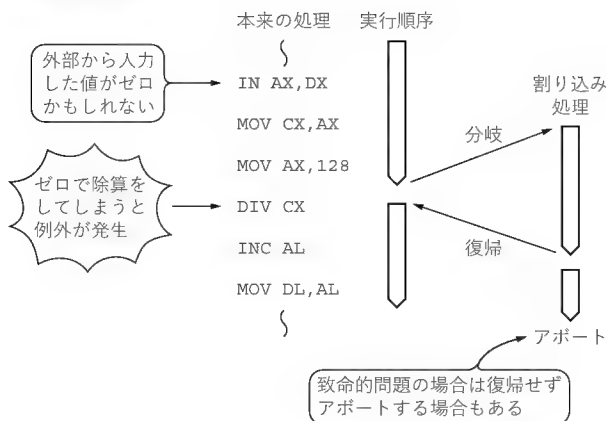
MPUでアプリケーションプログラムを実行する場合，通常は割り込みを意識しない．割り込みが発生すると，それまでの処理は中断され，特定の割り込み処理を行って元の処理に復帰する．アプリケーションプログラム側は割り込まれたことについて気付かない（**図1**）．MPUのプログラム実行順序としては，図のように一筆書き状の順番でプログラムを実行しているにすぎないが，人間の時間感覚で見ると，本来の処理と割り込み処理が平行に実行されたように見える．本来のプログラムが気付かないうちに並行動作が行われる……ここに割り込みの本質がある．

● ハードウェア割り込みとソフトウェア割り込み

割り込みは大きく分けて，MPUに接続された外部のデバイスが要求する**ハードウェア割り込み**と，プログラムで明示的に要求する**ソフトウェア割り込み**の二つがある．

ハードウェア割り込みとは，まさに**図1**のように，外部からの要因でジャンプ命令もないのにプログラムの実行が分岐することである．ハードウェア割り込みは，アプリケーションプログラムには見えない．外部のハードウェアの状態が変わったことを検出し，それにしたがって処理が必要な場合に利用する．一般的に，外部割り込みはMPUの処理とは非同期に行われる．

一方ソフトウェア割り込みは，割り込み処理へ切り替える命令をアプリケーションプログラム中に明示的に記述する（**図2**）．この意味で，ソフトウェア割り込みはサブルーチンコールのようにも見える．たいていのMPUには，ソフトウェア割り込み

〔図1〕割り込み処理の概念（ハードウェア割り込み）

〔図2〕ソフトウェア割り込み

〔図3〕例外の概念

を発生させるためのトラップ命令やシステムコール命令が用意されている．

● 例外とは何か

一般的に割り込みはプログラムの実行とは無関係(非同期)に発生するが，プログラムの実行結果によって発生する予期しない事象がある．たとえば，ゼロ除算，オーバフロー，アドレスエラー，ページフォールト(TLBミス)などがある．これらの発生によってもプログラムの処理は中断され，それら予期しない状態を処理するプログラムが実行される(**図3**)．これらは，要因がプログラムの実行そのものにあり，外部からの要因によって割り込まれたわけではないので，とくに**例外**と呼ぶ．

「例外」を辞書で引くと「通例の原則にあてはまらないこと．一般の原則の適用を受けないこと．また，そのもの．」とある．コンピュータの世界でもイメージは同じだが，命令の処理が通常と同じようには終了しない事象を表す．

どのような事象が発生したときに例外となるのかは，MPUのアーキテクチャによって異なる．たとえば，定義されていない命令コードを実行すると，あるアーキテクチャでは例外となるが，あるアーキテクチャではNOPと同じ動作となり，そのままプログラムを実行し続ける．

● 割り込みと例外の区別

要因発生後の動作，つまり割り込み処理へ分岐する動作は，割り込みも例外も共通である．しかし，割り込みの場合は元のプログラムに復帰するのが前提であるが，例外は場合によっては，致命的な事象と判断してプログラム処理を中止(アボート)することもある．

事象発生後の挙動が同じという点で，割り込みと例外は言葉の上での区別のみにも思える．実際，割り込みと例外を同一視するアーキテクチャのMPUも多い．そのような場合，外的要因によるハードウェア割り込みを**外部割り込み**，内的要因による例外とソフトウェア割り込みを**内部割り込み**と呼んで区別する．

割り込みと呼ぶか例外と呼ぶかは，そのMPUのアーキテクチャ上の決め事である．ここでは原則として，外的要因によるものを割り込み，内的要因によるものを例外として話を進める(とはいえ「ソフトウェア例外」とは呼ばないが……)．

● ベクタとハンドラ

割り込みが発生すると割り込み処理へ分岐するわけだが，どこに分岐するかを示すものを**割り込みベクタ**と呼ぶ．そして割り込み処理ルーチンのことを，**割り込みハンドラ**と呼ぶ．また割り込みと呼ぶか例外と呼ぶかに対応して，ベクタとハンドラも，割り込みベクタ，割り込みハンドラ，例外ベクタ，例外ハンドラと呼ばれる．

なお，後述する割り込みコントローラが与える割り込み番号も割り込みベクタと呼ぶが，割り込みの処理プログラム(割り込みハンドラ)の先頭アドレスも割り込みベクタと呼ばれる．この二つは別ものなので，混同しないようにしてほしい．ここではとくに断らない限り，割り込みハンドラの先頭アドレスという意味で割り込みベクタという言葉を使う．

● 割り込みベクタテーブル

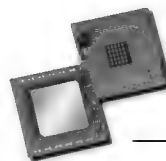
CISC系MPUの多くは，割り込みや例外に対する割り込みベクタの値，つまり割り込みハンドラのアドレスを自由に設定することができる．その割り込みベクタをある決められた順序でメモリ上に並べたものを**割り込みベクタテーブル**と呼ぶ．

多くの場合，割り込みベクタテーブルのベースアドレス，つまり先頭の割り込みベクタが格納されているアドレスは物理アドレスの0番地である．MMUをサポートするMPUでは，この割り込みベクタのベースアドレス(物理アドレスで指定する)を変更することが可能な場合が多い．そのため，割り込みベクタのベースアドレスを保持する特別なレジスタが用意されている．このベースアドレスレジスタの値を変更することで，割り込みベクタテーブルを任意のアドレスに配置することができる(**図4**)．

一方，RISC系MPUの多くは，割り込みベクタの値がアーキテクチャで一意に決められているので，割り込みベクタテーブルというものは存在しないことが多い．さらに，割り込みベクタの値は仮想アドレスだが，対応する物理アドレスは1対1で決まっている(たとえば，アドレス変換されない)ことが多い．

● 割り込みの受け付け，NMIとリセット

割り込みとは，本来の処理の途中で別の処理を行わせることだが，処理の内容によっては，実際に連続して実行しないと意味をなさない，途中で割り込み処理が実行されては都合の悪い場合もある．そのような場合は割り込み受け付けを禁止することもできる．

しかし外的要因の中には，非常に緊急性を有する事象もある．もしそれが発生した場合は，割り込まれると都合の悪い処理中でも，その緊急の割り込み処理を実行する必要があるだろう．このような重要な割り込みは，割り込み受け付け禁止ができない割り込みとして**ノンマスカブル割り込み**(Non Maskable Interrupt，NMI)を使う．通常，割り込みと呼ぶ場合は，ソフ

トウェアで割り込み受け付けを禁止することができる**マスカブル割り込み**のことを呼ぶ．

MPUのアーキテクチャによっては，リセットも割り込みもしくは例外に分類するものがある．割り込みベクタがプログラマブルなMPUであっても，さすがにリセット時は特定のアドレスから実行を開始したり，特定アドレスのメモリを読み込み，その値をアドレスとして実行を開始する（リセットベクタ）．

また，ノンマスカブル割り込みという意味では，リセットもノンマスカブルな割り込みといえる．しかもNMIよりも優先度が高く，MPUの中ではもっとも優先度の高い割り込みといえる．

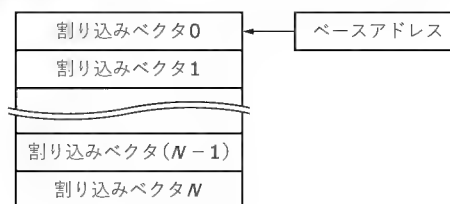
2 外部割り込みと例外の動作の概要

ここではハードウェア（外部）割り込みと例外の動作について解説する．以降ではとくに明記しない限り，ハードウェア割り込みを単に「割り込み」と示すことにする．ソフトウェア割り込みについては**コラム1**を参照してほしい．

● 割り込まれるプログラムの影響

割り込みや例外は，割り込まれるプログラム側からすれば意図しない場所で秘密裏に処理される．このときの動作はどうなっているのだろう．まずは，プログラムの実行を規定する要因を考える．ある瞬間のプログラムを完全に再現するには，

- プログラムの命令コードとデータ
- プログラムでアクセス可能なすべてのレジスタの値

〔図4〕割り込みベクタテーブル

- PC（プログラムカウンタ）の値
- SR（ステータスレジスタ）の値

といったデータが一意に定まっていればよい．これらの情報をコンテキストと呼ぶ．PCとはいうまでもなく，現在実行している命令コードのアドレスである．SRとはPSW（Program Status Word）やPSR（Program Status Register）とも呼ばれ，条件分岐用の条件フラグや実行レベルが含まれる（x86でいうところのFLAGレジスタ）．

これらのうち，プログラムの命令コードとデータは，そのプログラムの実行が終了するまで，物理メモリまたは補助記憶上に存在しているので，とくに気にする必要はない．レジスタの値は壊されると困るので，割り込みハンドラでは，そこで使用するレジスタの値をスタックなどに退避しておき，例外ハンドラを抜けるときに退避しておいた値を書き戻してやればよい．

Column 1

ソフトウェア割り込みとサブルーチンコール

ここでは，ソフトウェア割り込みとサブルーチンコールについて考える．

ソフトウェア割り込みは，トラップ命令やシステムコール命令などの，プログラムで明示的に記述し積極的に発生させる割り込みである．ソフトウェア割り込みは，OSが提供するサービスを得るためのシステムコールのインターフェースとして利用される．意味的にはサブルーチンコールと大差ない．それではなぜ，ソフトウェア割り込みというわずらわしい（わけでもないが）手順を踏むのであろうか．それには少なくとも二つの理由がある．

一つは，実行レベルの問題である．WindowsやLinuxでは，ユーザープログラムは，MPUの提供するユーザーモードで実行されている．それに対してOS内部はカーネルモードで実行される．通常のサブルーチンコールでは現在の実行レベルを保持するので，ユーザープログラムからコールしたサブルーチンでは特権命令を実行できない．ソフトウェア割り込みによって，実行レベルを特権レベルに上げることができる．

二つ目は，コールするアドレスの問題である．WindowsやLinux上のユーザープログラムは，基本的にすべて仮想アドレス上で動作する．一方，OSのサービスルーチンの先頭アドレスは一意に決まっている．その先頭アドレスを明示的にユーザープログラムで指定するには，仮想アドレスがどの物理アドレスに変換されるのかを知る手段がない以上，一般には不可能である．割り込みベクタテーブルは，通常，システムに一つだけ存在するので，OSのサービスを割り込みハンドラで指定するようにすれば，すべてのタスクから同じOSのサービスルーチンを実行できてむだがない．

歴史的に見れば，保護やアドレス変換がない昔のMPUでは，システムコールがサブルーチンコールによって行われていた．これは仕方のないことである（というかそれ以外の方法はなかった）．しかし，比較的新しいところでは，OS/2でもシステムコールをサブルーチンコールで実現していた．その当時，すでにMS-DOSではシステムコールにINT命令を使用していたので，OS/2は先祖帰りといえなくもない．なぜ，そのようなしくみを採用したのか，IBMの見解を聞いてみたいものである．OS/2を動作させるMPUが，アドレス変換がまだ洗練されてなかった80286だったことが一因かもしれない．

おもしろいところでは，Windows CEや一部のLinuxのシステムでは，システムコールにアドレスエラーを利用している．MPUにはトラップ命令やシステムコール命令が用意されているのに，なぜこうなっているのかは謎である．

レジスタは割り込みハンドラで使用しないこともあるが，PCとSRの値は必ず変更される．つまり，PCとSRがプログラムの挙動を性格付ける．

結論として，各レジスタやPCとSRを割り込み処理の前に保存し，割り込み処理を終了した後で元に戻してやれば，割り込まれたプログラムは何も知らずに処理を継続することができる．

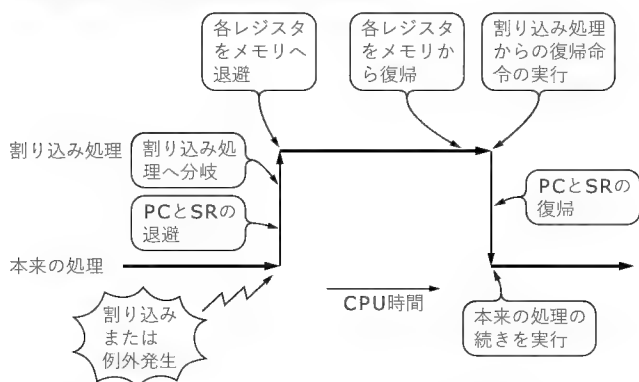
● 割り込み/例外発生時の動作

実際に割り込みや例外が発生したときのMPU内の動きについて見てみよう(**図5**)．多くのMPUでは，割り込みや例外が発生すると，PCとSRを自動的に特定の場所に退避するようになっている．また，外部から割り込みアクノリッジ(ベクタ)を読み込むMPUもある(詳細は実際のMPUでの動作の項目で説明)．

CISC系MPUでは，割り込みや例外が発生するとPCとSRを(割り込み用)スタックに退避し，割り込みからの復帰を指示する命令(RETIなど)を実行すると，スタックからPCとSRの元の値を取り出して，新たにPCとSRに設定しなおす．

RISC系MPUでは，スタックアクセス(＝メモリアクセス)を行うと処理速度が低下してしまうので，退避専用の特殊レジスタに値を格納する．割り込みハンドラの終了を指示する命令はPCとSRの値をこの特殊レジスタから取り出す．これらのレジスタは1組しか用意されていないのが普通で，多重に割り込みや例外が発生すると値が上書きされてしまう．多重に割り込みが発生する可能性がある場合は，スタックなりメモリなりに内容を退避する必要がある(RISCにもスタックという概念はある)．

割り込み発生前と割り込みハンドラからの復帰後で，プログラムで使用しているレジスタの値は保存されなければならない．このレジスタの退避/回復処理は，大量のメモリアクセスを伴うので，性能低下につながる．それを避けるため，アーキテクチャによっては，割り込みハンドラでのみアクセスできる，通常のレジスタとは独立なレジスタを提供していることもある．このような構造を**レジスタバンク**と呼ぶ．ARMなどのアーキテクチャは例外の種類ごとに数種類のレジスタバンクを備える．

また，割り込みからの復帰命令はMPUによって異なるが，だいたい次のような名称で呼ばれる．

RETI (RETurn from Interrupt)
RETE (RETurn form Exception)
IRET (Interrupt RETurn)
ERET (Exception RETurn)

この名称によって，そのMPUのアーキテクチャが割り込み/例外のことを，割り込み(Interrupt)と呼んでいるか例外(Exception)と呼んでいるかを知ることができる．

〔**図5**〕**割り込み/例外処理の動作の概要**

● 割り込み発生と割り込みマスク

一般的なMPUでは，一度に一つの割り込み要求しか受け付けないようにするため，割り込み発生時には新たな割り込みの受け付けが不可になる．ソフトウェアによる割り込みや例外処理中に発生する割り込みは，割り込み処理が終了するまで待たされる．具体的には，復帰命令を実行して割り込みが許可されるまで，新たな割り込みは受け付けない．

一方，ソフトウェアによる割り込みや例外処理中に発生する例外に関しては，禁止(マスク)する手段がない．多くの場合はその例外処理に移行するが，発生する例外の種類によっては2重例外による致命的例外となり，MPUの実行が停止する場合もある．

一般に，割り込みは例外処理中には受け付けが禁止されるが，意図的にSRを書き換えれば割り込みの受け付けを可能にすることもできる(多重割り込みについては後述)．

● 割り込み許可とマスク

通常，割り込みには許可ビットとマスクビットが用意されている．許可ビットとは割り込みの受け付けを許可するか否かを指定するビットである．割り込み発生時に新たな割り込みを受け付けないようにする機構は，この許可ビットを自動的に受け付け禁止に設定することで実現されていることが多い．

一方マスクビットとは，割り込みをマスク(覆い隠す＝禁止する)ためのビットである．MPUが割り込み端子を1本しかサポートしていない場合は，マスクビットの意味はない．許可ビットとまったく同じ意味となるからである．

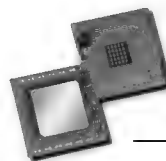
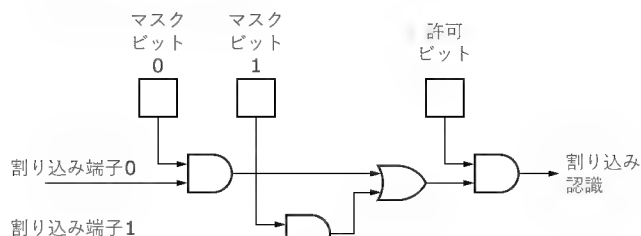
後述するように複数の割り込み入力がある場合，それぞれの割り込み要求に対して1対1にマスクビットが存在し，割り込み要因ごとに独立して割り込みを禁止する場合にマスクビットを使う．そしてMPUとして全割り込みの受け付けを許可するか否かを許可ビットで指定する．いずれにせよ，許可ビットとマスクビットの両方で割り込みが許可されていないと，割り込み要求は受け付けられない(**図6**)．

● 複数割り込みと優先順位

割り込み要求は大抵の場合，MPUの外部端子によって通知される．バスサイクルで与えられるMPUもあるが，ごく稀なケースなのでここでは割愛する．

MPUによっては外部割り込み入力が1本という場合もあるが，実際にシステムを構築する場合には，割り込み要因が10を越えることは珍しくなく，複数の外部割り込みを扱う要求が出

〔図6〕許可ビットとマスクビット

〔図7〕レベル方式による優先順位付き複数割り込み入力

てくる．割り込みが複数ある場合は，割り込みの優先順位をどうするかも問題である．

MPUによっては，外部割り込みを優先順位付きのレベルで識別できる．この場合，割り込み端子は複数本からなり，その端子状態が割り込み要求のレベルを表す．たとえば，割り込み端子の本数が3本なら，0～7の8種類のレベルを要求できる．このレベルはそのまま割り込みの優先順位となり，MPU内に記憶されている基準レベルと比較され，それより優先順位が高い場合は割り込みを受け付ける．

図7は，要求レベルの値が大きいほど優先順位が高いものと仮定し，0の場合が割り込みなしの状態となっているときの，割り込みを認識するしくみである．この基準レベルはソフトウェアで任意に変更できる．つまり，ある優先順位の割り込みを処理している場合は，それより優先順位の低い割り込み要求を受け付けないようにもできる．この場合，基準レベルが割り込みのマスクとして機能している．逆に，現在より優先順位の高い割り込み要求が発生すると受け付けてしまう．それを防ぐためには，ソフトウェアで現在処理中の優先順位を最高位に上げておく．

● 割り込みコントローラ

図7のようなMPUでは，割り込み要因に対応した値(レベル)を割り込み入力端子に入力する必要があるが，外部割り込みを発生させる一般的な外部デバイスは，割り込み要求時に割り込み出力端子をアサートする機能しかもたず，それ自身ではレベルを生成することができないものが多い．そのような場合は，プライオリティエンコーダ(優先順位の符号化器)を使用し，割り込み要因に対応したレベルをMPUに入力できるようにする．また複数の割り込みが同時に発生した場合は，もっとも優先順位の高い割り込み要因のレベルをMPUに入力する(**図8**)．このような，複数の割り込み要因を優先順位を考慮してMPUに伝達するデバイスを，**割り込みコントローラ**と呼ぶ．

MPUに割り込み端子が複数あっても，レベル入力方式でない場合もある．その場合は各割り込み端子自体が優先順位をもっている．たとえば，INT0，INT1，INT2という割り込み端子があれば，INT0 ＜ INT1 ＜ INT2の順に優先順位が高く，複数の割り込み端子が同時にアサートされる場合は，より高い優先順位の割り込みが受け付けられる．MPUに用意されている割り込み入力本数では足りない場合は，外部に割り込みコントローラをカスケード接続するなどして割り込み入力を拡張する必要がある(**図9**)．

またMPUによっては，複数の割り込み端子を有していても，それらにハードウェア的な優先順位がないこともある．その場合，マスカブル割り込みの割り込みベクタは1種類で，あとはソフトウェアで「よきに計らえ」ということになる．具体的には，すべての割り込み端子の状態がソフトウェアから見えるようになっていて，それを見ながらソフトウェアで適当に優先順位をつけて処理することになる．この場合，割り込みを認識するソフトウェアのステップ数が増加するので，割り込みハンドラの処理が重くなる．しかし，ハードウェア構成が単純なので，RISC系のMPUではこの構成がしばしば採用される．

● 多重割り込み

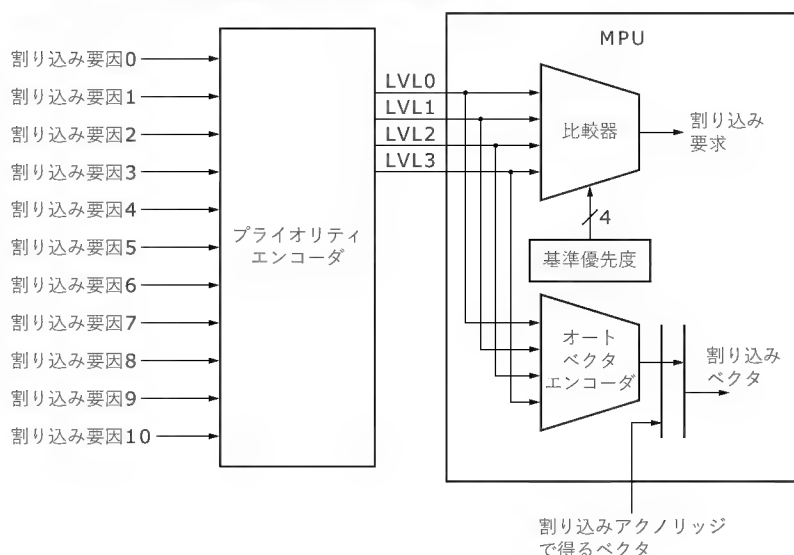
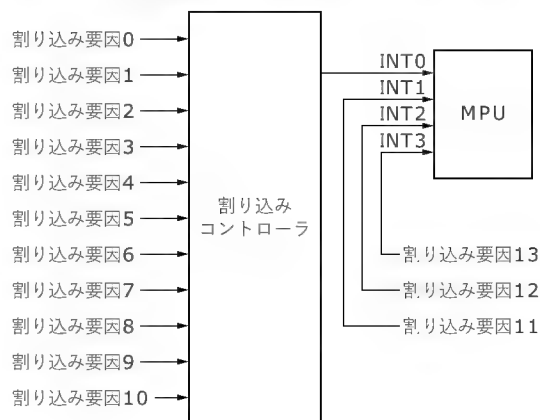
複数の割り込み要因が優先順位付きでMPUに入力される場合，優先順位の低い割り込み処理中に，より優先順位の高い割り込みが発生する可能性がある．

割り込み処理中(割り込みハンドラを実行中)は通常，新たな割り込みの受け付けはマスクされる．しかし割り込みハンドラ内でも，割り込み許可ビットをセットして，より優先度の高い割り込み要求の受け付けを許すようにもできる．これにより，より優先度が高い割り込みが発生した場合，そちらの割り込み処理を開始することができる．これを**多重割り込み**と呼ぶ．

多重割り込みは，CISC系MPUなどのPCやSRがスタックに保存されるMPUでは，とくに考慮が必要な問題はない．しかしRISC系MPUなどPCとSRが専用レジスタに退避されるだけの方式では，多重割り込みを許可する前に，その専用レジスタの内容が書き潰されないように，元の値をスタックなどの領域に退避しておく必要がある．

● 割り込みを受け付けるタイミング

割り込み要求が発生したとき，MPUがその要求を受け付けるタイミングはいつだろうか．それは，MPUが割り込み処理を行

〔図8〕レベル方式による割り込み入力本数の拡張

〔図9〕割り込み入力端子に優先順位がある場合の割り込み入力の拡張

うのに都合のいいタイミングである．いくらなんでも，命令実行の途中（具体的には結果をデスティネーションレジスタにライトバックする前に）で割り込みを受け付けたりしたら，一時的に保持している値が壊れてしまうので，正しい結果をライトバックできない．この意味で，割り込みは命令の実行終了後，次の命令の実行前のタイミングで受け付けられるのが普通である．

RISCでは1命令の実行時間は基本的に1クロックなので，たいていの場合は，割り込みを要求してから1クロック後には割り込みが受け付けられる．ただし，FPUの除算命令などは実行に50クロック以上もかかることもあり，その場合には，割り込みを受け付けるまでに最悪50クロック程度かかることになる．

CISCの場合，1命令で行う処理の複雑さゆえ，命令の実行時間は通常1クロック以上である．たとえば，文字列転送命令や，倍精度の浮動小数点命令の実行には200クロック以上かかることも珍しくない．

さすがにこれでは割り込み応答性のよいリアルタイムOSを作ることは難しい．そこで，CISCのMPUでは，実行時間の長い命令に関しては，例外的に，命令実行の途中で割り込みを受け付けるようになっている．割り込み発生時にスタックに積まれるPCの値は，一般には，次の命令のPC（Next PC）であるが，命令実行中に割り込みを受け付ける場合は，実行を中断した命令（実行中の命令）のPC（Current PC）である．このため，割り込みハンドラでRETIなどの復帰命令を実行すると，中断した命令から実行が再開される．MPU内部では命令の再開処理がうまくいくようなしくみが用意されているのである．

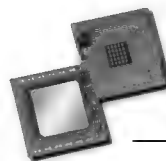
● 割り込み機能の実装

実際にMPUで割り込み機能を実装するためにはどうするのだろうか．簡単に説明すると，命令がパイプラインを流れる間に割り込みを受け付けると，その命令を割り込みベクタへ分岐するジャンプ命令に置き換える．割り込みと例外はほとんど同じ処理になるので，割り込みがあるかないかを調べるタイミング（サンプリングという）は例外検出を行うタイミングと同じ場合が多い．つまり，例外発生時も，その命令を例外ベクタへのジャンプ命令に置き換えることで実現できる．

たとえば，IF（命令フェッチ），RF（デコード），EX（実行），DC（データアクセス），WB（ライトバック）からなる5段パイプラインのMPUを考える．例外として考えられるのは，RFステージでの未定義命令例外（ブレークポイントやシステムコールを含む）とDCステージでのアドレスエラー（データのTLBミスを含む）やEXステージの結果に依存するトラップやオーバフローなどがある．このような場合，RFステージかDCステージで割り込みをサンプリングするのが普通である．割り込み応答を良くしたい場合はRFステージとDCステージの両方で割り込みのサンプリングを行う．

ただし，DCステージで割り込みをサンプリングする場合は，命令のデコードはすでに終了しているので，単純に例外ベクタへのジャンプ命令に置き換えることはできない．この場合は，その命令をジャンプ命令に置き換えるというよりは，次にフェッチする命令をジャンプ命令に置き換えると考える．処理的にはRFステージでサンプリングするよりも複雑である（それならばすべてDCステージでサンプリングすればいいという考えも当然ある）．

図10に，RFステージで割り込みをサンプリングする場合の割り込み処理の概略を示す．パイプラインの制御はジャンプ命令と同じでよいので，割り込みを受け付けた後続命令を無効化する処理もジャンプ命令と同様の制御で実現できる．

割り込みだけでなく，例外処理も同じ実装でよいが，命令フェッチ時のアドレスエラーやTLBミスの場合は，例外ベクタ

へのジャンプ命令をフェッチしてくると思えばよい．

とくに例外は，実行（EXステージ）が終わらないと発生の有無がわからない場合もあるので，DCステージでのサンプリングは必須である．DCステージは演算結果を書き戻す（WBステージ）直前であり，無効な結果を書き戻さないようにするための最後のチャンスである（割り込みなら1命令後で発生してもかまわない）．DCステージで例外を検出した場合はWBステージでの書き込みを禁止して，次にフェッチする命令をジャンプ命令に置き換える．

〔図10〕割り込み機能の実現

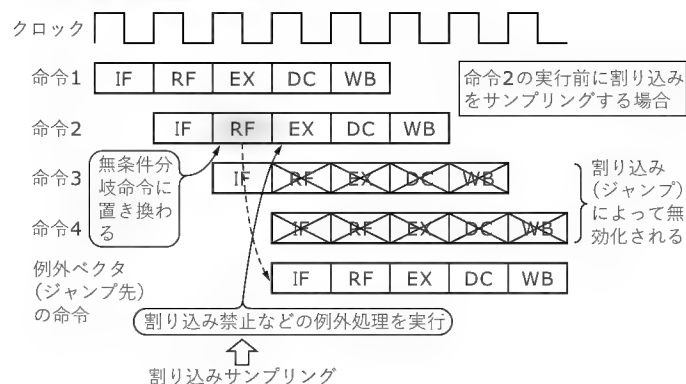
3 割り込みと例外処理の実際

それでは，実際のMPUにおける割り込みと例外処理について，いくつかのアーキテクチャのMPUを取り上げて説明する．

● x86の場合

x86での割り込みはハードウェア割り込みのことを指し，周辺デバイスからの割り込み要求によって発生する．例外は，トラップ，フォールト，アボートに区別される．トラップとはINT xといったソフトウェア割り込み，フォールトは主としてMMU関連の例外，アボートは処理が続けられないようなエラー発生時の例外である．

▶ハードウェア構成

x86アーキテクチャのMPUでは，割り込みコントローラとしてインテルの8259というLSIを想定している（最近ではAPIC＝Advanced Programmable Interrupt Controllerがその役割を果たす）．MPUと割り込みコントローラは**図11**（**a**）のように接続される．8259は1個で八つまでの割り込み要因しか処理できない．それ以上の割り込み要因が必要な場合は，**図11**（**b**）のように割り込みコントローラをカスケード接続して対応する．

割り込み発生時の割り込みコントローラの動作を**図12**に示す．これを割り込みアクノリッジサイクルと呼ぶ．8259は割り込みを出力するデバイスからの割り込み要求を察知すると，MPUの外部端子である割り込み要求端子（INTR）をアサートし，外部割り込み要求の存在を知らせる．MPUは外部割り込みの存在を感知すると，割り込みアクノリッジを示す信号を（S2～S0端子）出力する．そこで，割り込みコントローラはデータバスに割り込み番号（ベクタ）を与えて割り込みの種類を示す．割り込みアクノリッジサイクルが2回発生するのは8259の都合である．1回

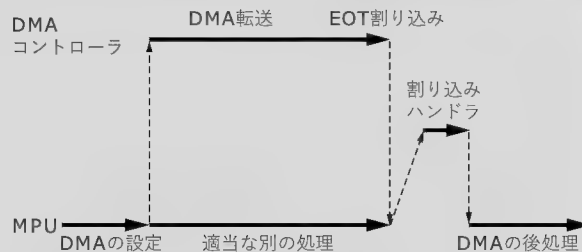
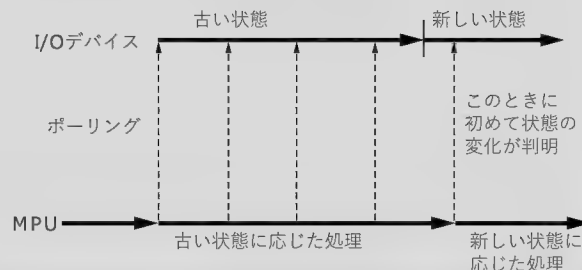
Column 2

割り込みとポーリング

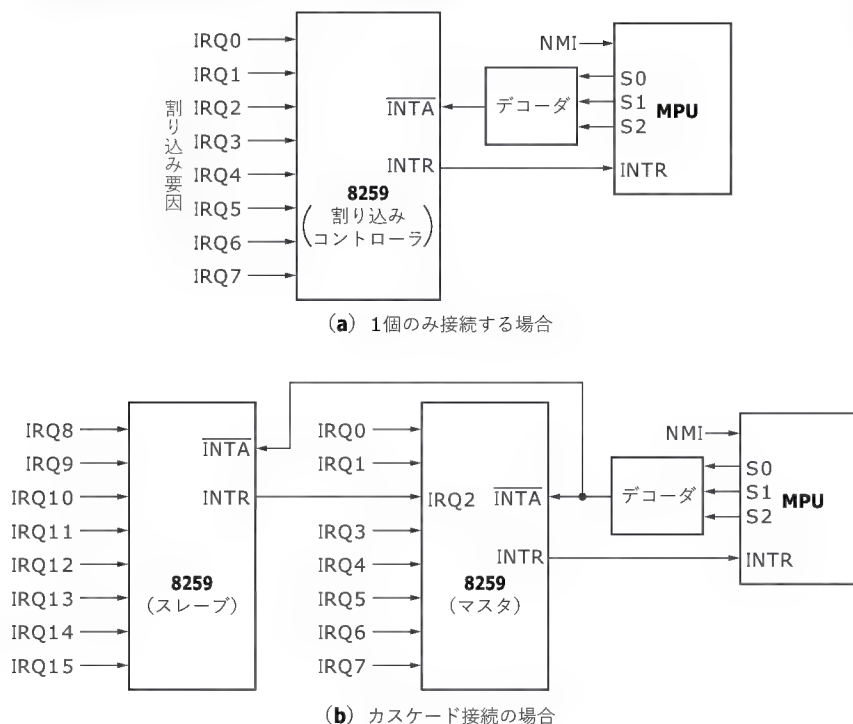
割り込みの利点の一つとして，ある処理の終了を割り込みで通知するようにしておけば，その間に別の処理を並行して実行できることが挙げられる．たとえば，DMAの待ち合わせに割り込みが多用される．DMAコントローラの多くは転送終了時にEOT（End Of Transfer）またはTC（Terminal Count）といった割り込みを発生する．DMA転送を割り込みで待ち合わせる処理のイメージを**図A**に示す．

このような，動作の終了で割り込みを出力する機能をもたないデバイスによる処理の待ち合わせには，そのデバイス内のステータスを定期的にチェックして，処理が終了したかを判定しなければならない（**図B**）．このように，定期的にステータスの状態をチェックすることを**ポーリング**という．ポーリングは，ソフトウェアによる単純なループであることも少なくない．ポーリングによる処理の待ち合わせは，状態の変化を知るまでに遅れが生じるので，割り込みと比べると効率が悪い．また，その間に別の処理を行えないという点でもポーリングの効率は悪い．

〔図A〕DMA転送終了割り込みによる処理の待ち合わせ

DMAコントローラ　DMA転送　EOT割り込み　割り込みハンドラ　MPU　DMAの設定　適当な別の処理　DMAの後処理

〔図B〕ポーリングによる処理の待ち合わせ

〔図11〕x86用割り込みコントローラ8259の接続

(a) 1個のみ接続する場合

(b) カスケード接続の場合

〔図12〕x86の割り込みアクノリッジサイクルの動作

〔表1〕x86の割り込み番号とその要因

割り込み番号	種　類	要　因
0x00	フォールト	除算エラー
0x01	フォールト	デバッガ割り込み(トレース)
0x02	アボート	NMI
0x03	トラップ	INT 3(ブレークポイント)
0x04	トラップ	INTO
0x05	フォールト	配列境界違反
0x06	フォールト	無効命令
0x07	フォールト	コプロセッサ無効
0x08	アボート	ダブルフォールト
0x09	アボート	コプロセッサセグメントオーバラン
0x0A	フォールト	無効TSS
0x0B	フォールト	セグメント不在
0x0C	フォールト	スタック例外
0x0D	フォールト	一般保護例外
0x0E	フォールト	ページフォールト
0x10	フォールト	コプロセッサエラー
0x11	フォールト	アラインメントチェック
0x12	アボート	マシンチェック
0x13	フォールト	ストリーミングSIMD拡張
0x12～0x1F		予約済み(使用不可)
0x20～0xFF		ユーザー用(外部割り込み/INT命令)

目で割り込みが発生したことを認識し，2回目で割り込みベクタを返す．なおここでいうベクタとは，割り込みハンドラの先頭アドレスではない点に注意してほしい．

割り込みアクノリッジサイクル自体は「要求された割り込みを受け付けた」という意味ももっている．割り込みアクノリッジが発生しないということは，要求された割り込みが無視されたということである．これは割り込みがマスク(禁止)されている場合に起こり得る．その場合，割り込みコントローラは割り込み要求端子をアサートし続け，割り込みアクノリッジが発生するのを待つのが普通である．通常，割り込み要求端子は割り込みアクノリッジが発生するまでアサートし続ける．

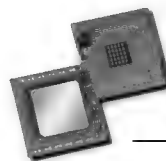
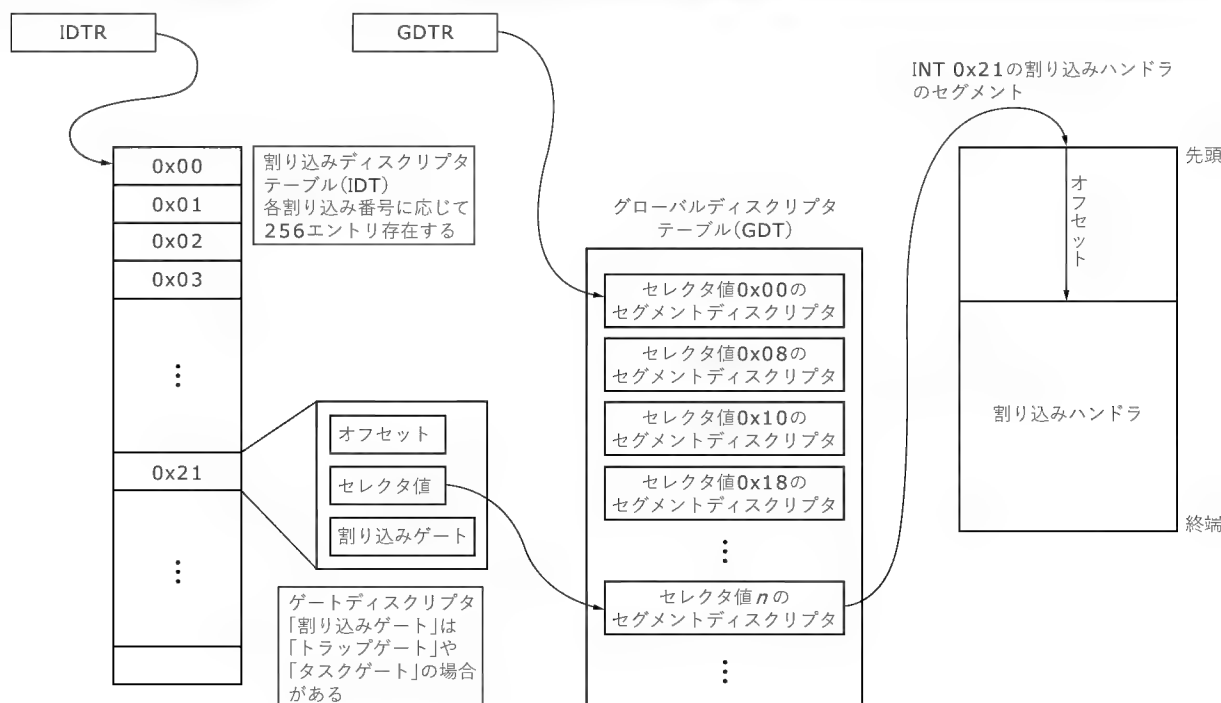
▶割り込み番号とその要因

x86がサポートする割り込み番号とその要因を**表1**に示す．ソフトウェア割り込みを発生するINT命令はパラメータとして0～255の割り込み番号を指定することができる．このため，INT命令によってすべての割り込み/例外を発生させせることが(理論上)可能である．外部割り込みの割り込み番号は，先ほど説明した割り込みコントローラから与えられる．

▶リアルモードでの動作

割り込み/例外処理の挙動はリアルモードとプロテクトモードで異なる．

リアルモードでは，0x00000番地から始まる256エントリの割り込みベクタテーブルで，割り込み/例外の割り込み番号とその処理ハンドラのアドレスが対応付けられる．割り込みベクタテーブルの各エントリは，2バイトのオフセットアドレスと2バイトのセグメントアドレスから構成される．

割り込み/例外が発生すると，MPUはフラグレジスタ，CSレジスタ，IPレジスタをスタックにプッシュして例外スタックフレームを作り，発生した割り込み/例外の割り込み番号に対応する割り込みベクタテーブルのエントリからオフセットアド

〔図13〕割り込みハンドラの選択(プロテクトモード)

レスとセグメントアドレスを読み出し，それぞれの値をIPレジスタ，CSレジスタに設定することにより，処理ハンドラに分岐する．

▶プロテクトモードでの動作

プロテクトモードの場合は，割り込みベクタテーブルではなく，割り込みディスクリプタテーブル(IDT)が使用される．割り込みディスクリプタテーブルの先頭アドレスは，0x00000番地に固定ではなく，IDTRレジスタによって設定される．割り込みディスクリプタテーブルは割り込み番号とその処理ハンドラのアドレスを決定するゲートディスクリプタとを対応付ける256エントリのテーブルである．ゲートディスクリプタは2バイトのセレクタ値，4バイトのオフセットアドレス，1バイトのスタックコピーカウント，1バイトのゲートの種類から構成される8バイトのデータである．

大ざっぱにいえば，リアルモードでの割り込みベクタテーブルのエントリに対して，オフセットアドレスが2バイトから4バイトに拡張されたと思えばよい．そして，セレクタ値が間接的にセグメントの先頭アドレスを指し示す．

プロテクトモードにおいて割り込み/例外が発生すると，スタックポインタが特権レベル0のスタックポインタに切り替わる．そして，その新しいスタックに，古いスタックポインタ(SS：ESP)をプッシュし，その後，EFLAGSレジスタとCSレジスタとEIPレジスタの値をプッシュして，ゲートディスクリプタで指定された処理ハンドラに分岐する(**図13**)．割り込み/例外処理を行った後，IRET命令を実行すると，特権レベル0スタックからSS：ESP，EFLAGS，CS：EIPを回復する．

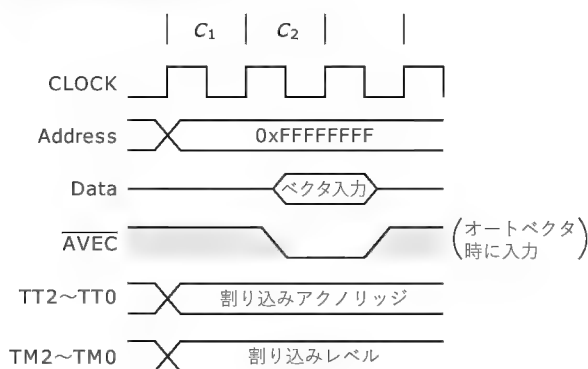
x86において割り込みと例外の挙動の差異は，処理ハンドラに分岐した時点で，新しいFLAGSレジスタまたはEFLAGSレジスタの割り込み許可ビットが禁止(割り込み発生時)になっているか，前の値を引き継いでいる(例外時)だけである．

● MC680x0の場合

▶ハードウェア構成

68000系では，割り込みコントローラを含む周辺デバイスとして，MC68901というMFP(Multi-Function Peripheral)が存在する．とくに組み込み制御用途のMPUでは専用の周辺デバイスが用意され，割り込みコントローラもそれに含まれていることが多い．割り込みコントローラは各社独自のASICとして供給されることもある．

図14に680x0での割り込みアクノリッジサイクルを示す．680x0での割り込みのベクタ番号は一定しておらず，MPU外部の割り込みコントローラによって与えられる．割り込みを受け付けると，MPUは割り込みアクノリッジバスサイクルを発行して，割り込みコントローラにベクタ番号を問い合わせる．割り込みコントローラは，発生している割り込みの種類に応じて，ベクタ番号(64～255)を返すか，オートベクタを使用する($\overline{\text{AVEC}}$端子をアサートする)かを決定する．オートベクタというのは，割り込みの優先レベル(1～7)に固定のベクタ番号である．具体的には優先レベルに24を加えた25～31がベクタ番号となる．オートベクタは，システムと密接した割り込み処理に利用されることが多いようである．

〔図14〕680x0の割り込みアクノリッジサイクル

〔表2〕680x0の例外ベクタテーブル

ベクタ番号	オフセット	割り当て
0 1 2 3	0x000 0x004 0x008 0x00C	リセット時割り込みスタックポインタ リセット時プログラムカウンタ アクセスフォールト アドレスエラー
4 5 6 7	0x010 0x014 0x018 0x01C	不正命令 整数ゼロ除算 CHK，CHK2命令 FTRAPcc，TRAPcc，TRAPV命令
8 9 10 11	0x020 0x024 0x028 0x02C	特権違反 トレース ライン1010エミュレータ （未実装Aライン命令コード） ライン1111エミュレータ （未実装Fライン命令コード）
12 13 14 15	0x030 0x034 0x038 0x03C	（予約） コプロセッサプロトコル違反 フォーマットエラー 未初期化割り込み
16～23	0x040～0x05C	（予約）
24 25 26 27	0x060 0x064 0x068 0x06C	スプリアス割り込み レベル1割り込みオートベクタ レベル2割り込みオートベクタ レベル3割り込みオートベクタ
28 29 30 31	0x070 0x074 0x078 0x07C	レベル4割り込みオートベクタ レベル5割り込みオートベクタ レベル6割り込みオートベクタ レベル7割り込みオートベクタ
32～47	0x080～0x0BC	TRAP #0～#15命令
48 49 50 51	0x0C0 0x0C4 0x0C8 0x0CC	FPアンオーダ状態での分岐または セット FP精度落ち FPゼロ除算 FPアンダフロー
52 53 54 55	0x0D0 0x0D4 0x0D8 0x0DC	FPオペランドエラー FPオーバフロー FPシグナリングNot a Number FP未実装データ形式
56 57 58	0x0E0 0x0E4 0x0E8	MMU構成エラー MC68851で使用 MC68851で使用
59～63	0x0EC～0x0FC	（予約）
64～255	0x100～0x3FC	ユーザー定義ベクタ

もし，割り込みアクノリッジバスサイクルに対して何も応答が返らない場合はスプリアス割り込みとなる．これは割り込みの要因が不明な割り込みで，MPUとしては処理する方法がわからない．通常のシステムでは，ノイズによる誤動作などとして，スプリアス割り込みは無視される（割り込みハンドラはRTEのみ）．

▶割り込み/例外の動作

MC680x0の割り込み/例外処理は，例外ベクタテーブルと例外スタックフレームを使用する．ベクタベースレジスタ（VBR）は256個の例外ベクタからなる1024バイトの例外ベクタテーブルの先頭アドレスを保持する．例外ベクタは，リセットベクタを除き，例外処理ルーチンの先頭アドレスである．**表2**に例外ベクタテーブルの内容を示す．このうち，リセットベクタはISP（割り込みスタックポインタ）の初期値とPCの初期値（実行開始アドレス）からなる．例外ベクタの格納されているアドレスは，例外の種類に応じてMPUが自動的に割り当てる8ビットのベクタ番号から決定される．また，いくつかの例外については外部デバイスが例外ベクタを供給する．例外ベクタアドレスは，例外ベクタを4倍し，VBRの値に加算して決定される．

割り込み処理はスーパバイザスタックに例外から復帰するための情報を積む．これらは，例外の種類によって異なる，例外スタックフレームと呼ばれる構造を採る．例外スタックフレームは，SR（ステータスレジスタ），PC（プログラムカウンタ），ベクタのオフセット，スタックフレームの形式を示す領域と，追加情報からなる．

例外/割り込み処理の後，RTE命令を実行すると例外スタックフレームからMPUの再実行に必要な情報が読み込まれて，実行を再開する．

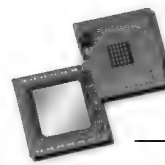
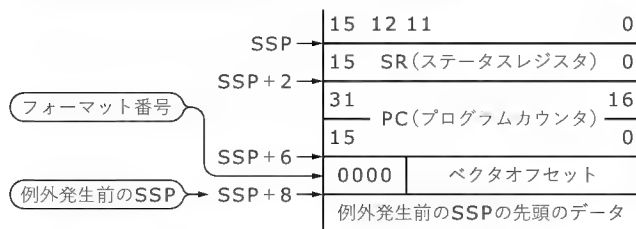
MC680x0で定義されている例外スタックフレーム（フォーマット0）を**図15**に示す．例外スタックフレームの種類は，MC68000，MC68010，……，MC68060と世代が進むごとに（対処的に？）拡張され，最終的には10種類を超えた．付け焼刃のようで，アーキテクチャとしてはあまり美しくない．

● ARMの場合

▶レジスタ構成

ARMのアーキテクチャでは，割り込み/例外発生時に，ユーザーレジスタの退避の必要性をなくすため，レジスタバンクが用意されている．このレジスタバンクは割り込み/例外の種類（モード）に応じて5バンクが独立に存在する．このレジスタバンクのレジスタの一部はユーザーモードのレジスタと共通になっていて，モード間で共通にアクセスできる（**図16**）．

多くのモードでは，R13とR14を固有にもっている．R13にはそのモードでのスタックポインタの値が格納され，R14には割り込み/例外からの復帰アドレスが自動的にセットされる．R14には割り込み/例外を発生した次の命令のアドレスが格納されるので，ユーザーモードへの復帰時には，処理モードに応じて，適当な値をR14から減算してPCに格納する．

高速割り込みモード（FIQ）では，コンテキストスイッチの

〔図15〕680x0の例外スタックフレームの構造(フォーマット0)

例外の種類	PCが指す位置
割り込み	次の命令
フォーマットエラー	RTE命令またはFRESTORE命令
TRAP #N	次の命令
不当命令	不当命令
Aライン命令	Aライン命令
Fライン命令	Fライン命令
特権違反	特権違反を発生させた命令の最初のワード
浮動小数点命令実行前	浮動小数点命令
未実装整数	未実装整数命令
未実装実効アドレス	未実装実効アドレスを使用した命令

〔図16〕ARMのレジスタ構成

モード						
ユーザーモード	特権モード					
		例外モード				
ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ

◆ 汎用レジスタ

ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8
R9						R9
R10						R10
R11						R11
R12						R12
R13(SP)		R13	R13	R13	R13	R13
R14(LR)		R14	R14	R14	R14	R14
R15(PC)						

◆ ステータスレジスタ

ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ
CPSR						
		SPSR	SPSR	SPSR	SPSR	SPSR

オーバヘッドを軽減するため，R8～R14がモード固有のレジスタとして用意されている．例外スタックフレームは存在しない．その代わり，ステータスレジスタは，新しいレジスタバンクに存在するSPSRに退避される．PCは新しいレジスタバンクのR14に退避される．

▶割り込み/例外の動作

割り込み/例外発生時のMPUの動作は，次のとおりである．なおARMでは，ベクタアドレスは一つについて4バイトの領域しかないので，通常は処理ルーチンへの分岐命令が格納されている．(6)と(7)がソフトウェアによる処理である．

(1) 例外に対応する処理モードに移行
(2) 戻りアドレスを新しい処理モードのレジスタバンクのR14に退避
(3) CPSRの値を新しい処理モードのレジスタバンクのSPSRに退避
(4) CPSRの所定ビットをセットして外部割り込み不可にする
(5) 処理モードに応じた例外ベクタアドレス(**表3**)へ分岐する
(6) 例外処理を実行
(7) ソフトウェア割り込み，未定義命令トラップからの復帰時

➡MOVS　PC, R14　　(R14をPCに格納)

IRQ, FIQ, プリフェッチアボートからの復帰時

➡SUBUS　PC, R14, #4 (R14から4を減算してPCに格納)

データアボートからの復帰

➡SUBUS　PC, R14, #8 (R14から8を減算してPCに格納)

(命令の最後のSは同時にSPSRをCPSRに回復することを意味する)

なお，多重割り込みを行っている場合は，R14(戻りアドレスの基準)がスタックにある．この場合は多重レジスタ転送命令の

〔表3〕ARMの例外ベクタアドレス

割り込み/例外の種類	モード	ベクタアドレス
リセット	SVC(スーパバイザ)	0x00000000
未定義命令	UND(未定義)	0x00000004
ソフトウェア割り込み	SVC(スーパバイザ)	0x00000008
命令フェッチメモリフォールト	Abort(アボート)	0x0000000C
データアクセスメモリフォールト	Abort(アボート)	0x00000010
アドレス例外(26ビットアドレス)	Abort(アボート)	0x00000014
IRQ(通常の割り込み)	IRQ	0x00000018
FIQ(高速割り込み)	FIQ	0x0000001C

```
LDMIA  R13!, {R0-R3, PC}^
```

によって，例外からの復帰ができる(同時にCPSRを回復する)．R13はスタックポインタであり，作業用レジスタとして使われるR0～R3がスタックに退避されている場合を示している．レジスタリストの終わりの^が，CPSRを同時に回復することを指定する．

▶最新アーキテクチャでは割り込み機構を高速化

ARMはv6アーキテクチャで，例外/割り込み処理の高速化をめざしている．具体的には，

- 新しい割り込みスタック機構(SRS, RFE命令)
- 命令によるモード変更(CPSIE, CPSID命令)
- 発生順序を規定しないアボートをサポート
- 低レイテンシモードの採用(実装依存)
- ベクタ割り込みモードをサポート

である．

● MIPSの場合

▶ハードウェア構成

MIPS系のMPUは，通常5本の(マスカブル)割り込み端子をもっているが，それらの間に優先順位はない．すべてソフトウェアでの処理にまかされている．また，割り込みを受け付けても割り込みアクノリッジサイクルは発行しない．さらに，割り込みは端子の状態が原因レジスタの特定のフィールドにそのまま見えているだけなので，割り込みを確実に認識するためには，割り込み処理が終了するまで割り込み端子の状態を保持する必要がある．

通常のMPUでは割り込みアクノリッジサイクルが発行されると，割り込み要求を取り下げてよい(その割り込みは受け付けられたことが保証される)．MIPSでは，特定のI/Oポートにアクセスしたら割り込み要求を取り下げるというしくみを，外部回路で実現しなければならない．

▶割り込み/例外の動作

MIPSの割り込み例外処理は単純である．ほとんどすべての例外は共通のベクタアドレスへ分岐する．例外スタックフレームは存在せず，ステータスレジスタは例外ビット(EXLまたはERL)がセットされることで特権レベルに移行したことを示す．一方，PCは特定の特権レジスタ(EPCまたはErrorEPC)に退避される．割り込み/例外の要因は，ほとんどの場合同じアドレス(共通例外ベクタという)に分岐するので，原因レジスタに格納される例外コードを読み出して区別する．**表4**に，原因レジスタに格納される例外コードを示す．

割り込み/例外発生時のMPUの動作は，次のとおりである．(1)～(3)は外部割り込みの場合である．例外発生時は直接(4)に移行する．(7)～(9)がソフトウェアによる処理である．

(1) 割り込み要求が発生(INT0～INT4)
(2) INT0～INT4端子の状態とSRのマスクビット(IM0～IM4)の論理積(AND)が原因レジスタの割り込み保留領域(IP0～IP4)に反映される
(3) IP0～IP4のどれか一つが1であり，かつSRの割り込み許可ビット(IE)が1なら割り込みが発生する
(4) カーネルモード(相当)に移行する(EXLまたはERLが1)．同時に割り込み不可になる(EXLやERLが1のときは割り込み不可)
(5) 戻りアドレスを特定の特権レジスタ(EPCまたはErrorEPC)に退避
(6) 発生要因に応じた例外ベクタアドレス(**表5**)へ分岐する
(7) 外部割り込みの場合は割り込みの要因を取り下げる
(8) 割り込み処理を行う
(9) ERET命令を実行する
(10) EXL＝1の場合はEPCのアドレスに分岐しEXLを0にする．ERL＝1の場合はErrorEPCのアドレスに分岐しERLを0にする

〔表4〕MIPS系の原因レジスタの例外コード

例外コード	略号	説明
0	Int	割り込み
1	Mod	TLB変更例外
2	TLBL	TLB不一致例外(ロード，命令フェッチ)
3	TLBS	TLB不一致例外(ストア)
4	AdEL	アドレスエラー(ロード，命令フェッチ)
5	AdES	アドレスエラー(ストア)
6	IBE	バスエラー(命令フェッチ)
7	DBE	バスエラー(ロード，ストア)
8	Sys	システムコール
9	Bp	ブレークポイント
10	RI	予約済み命令例外
11	CpU	コプロセッサ使用不可例外
12	Ov	演算オーバフロー例外
13	Tr	トラップ
14	VCEI	命令仮想コヒーレンシ例外
15	FPE	浮動小数点演算例外
16～22	未使用	
23	WATCH	ウォッチ例外
24～30	未使用	
31	VCED	データ仮想コヒーレンシ例外

〔表5〕MIPS系の例外ベクタアドレス

例外・割り込みの種類	アドレス
リセット・NMI	0xFFFFFFFFBFC00000
キャッシュエラー	0xFFFFFFFFA0000100(BEV=0) 0xFFFFFFFFBFC00300(BEV=1)
TLB不一致(ミス)(EXL=0)	0xFFFFFFFF80000000(BEV=0) 0xFFFFFFFFBFC00200(BEV=1)
XTLB不一致(ミス)(EXL=0)	0xFFFFFFFF80000080(BEV=0) 0xFFFFFFFFBFC00280(BEV=1)
その他	0xFFFFFFFF80000180(BEV=0) 0xFFFFFFFFBFC00380(BEV=1)

例外ベクタアドレスは，リセット直後，ステータスレジスタのBEVビットをクリアするまでと，BEVビットをクリアした後で異なる．BEVとはBootstrap Exception Vectorの略で，まだ，キャッシュやTLBを初期化する前の状態を表す．ソフトウェアではそれらの初期化後にBEVビットを0にクリアすることが要請されている．このため，BEVが1の間は非キャッシュで非マップ(アドレス変換されない)領域が例外ベクタになっている．

▶最新アーキテクチャでは割り込み機構を高速化

MIPSの割り込み方式は単純でわかりやすいが，反面，高速な処理には適さない．そこでMIPS社は，2001年10月に発表した拡張機能で，割り込み処理を高速化する機構を強化した．詳細は不明だが，ARMと同様なレジスタバンクを16組もち，割り込みの種類に応じて16種の割り込みベクタを生成するアーキテクチャになるという．

● SH(SuperH)の場合

▶ハードウェア構成

SHの割り込みは，4ビットの優先順位(レベル)方式を採用している．

SH-1/SH-2では8本の外部割り込み端子(IRQ0～IRQ7)と内

〔表6〕SH-1/SH-2の例外ベクタ

例外要因	ベクタ番号	ベクタテーブル
パワーオンリセット PC SP	0 1	0x00000000 0x00000004
マニュアルリセット PC SP	2 3	0x00000008 0x0000000C
一般不当命令	4	0x00000010
(システム予約)	5	0x00000014
スロット不当命令	6	0x00000018
(システム予約)	7 8	0x0000001C 0x00000020
CPUアドレスエラー	9	0x00000024
DMAアドレスエラー	10	0x00000028
割り込み NMI USER BREAK	11 12	0x0000002C 0x00000030
(システム予約)	13 : 31	0x00000034 : 0x0000007C
トラップ命令 (ユーザーベクタ)	32 : 63	0x00000080 : 0x000000FC
割り込み IRQ0 IRQ1 IRQ2 IRQ3 IRQ4 IRQ5 IRQ6 IRQ7 内蔵周辺 : 内蔵周辺	64 65 66 67 68 69 70 71 72 : 255	0x00000100 0x00000104 0x00000108 0x0000010C 0x00000110 0x00000114 0x00000118 0x0000011C 0x00000120 : 0x000003FC

蔵する周辺ユニットからの割り込みが，MPUへの割り込み要因となる．これらの割り込み要因は，5本の割り込み優先順位レジスタ(IPR)で独立に優先順位を指定することができる．いずれかの割り込みが要求されると，それに対応した優先順位がMPUに入力される．

一方SH-3では，6本の外部割り込み端子(IRQ0～IRQ5)，16本のポート割り込み(PINT0～PINT15)，内蔵周辺ユニットからの割り込みに優先順位を与える方式のほか，4ビットの優先順位(IRL0～IRL3)を直接外部から入力することもできる．SH-4では，4ビットの優先順位入力(IRL0～IRL3)がユーザーに直接見えるようになっている．

いずれにしろ，割り込み要求(優先順位入力)が，ステータスレジスタ(SR)内の割り込みマスク領域(IMASK)の値よりも優先度が高いときに割り込みを受け付ける．割り込みマスク領域の初期値は最高の優先順位になっているので，MPUの初期化段階で適当な値をIMASKに設定する必要がある．

▶例外ベクタの構成

SHの例外ベクタの構成は，SH-2までとSH-3以降でまったく異なっている．SH-1/SH-2は例外要因それぞれに対して，0x00000000番地から始まる例外ベクタテーブルのオフセットが規定されている(**表6**)．この方式は，MC680x0の方式に酷似している．

〔表7〕SH-3/SH-4の例外ベクタ(抜粋)

例外要因	ベクタアドレス	例外要因
パワーオンリセット	0xA0000000	0x000
マニュアルリセット	0xA0000000	0x020
TLB多重ヒット	0xA0000000	0x140
リードアドレスエラー	VBR + 0x100	0x0E0
リードTLBミス	VBR + 0x400	0x040
リードTLB保護違反	VBR + 0x100	0x0A0
ライトアドレスエラー	VBR + 0x100	0x100
ライトTLBミス	VBR + 0x400	0x060
ライトTLB保護違反	VBR + 0x100	0x0C0
一般不当命令例外	VBR + 0x100	0x180
スロット不当命令例外	VBR + 0x100	0x1A0
初期ページ書き込み	VBR + 0x100	0x080
TRAPA命令	VBR + 0x100	0x160
USER BREAK TRAP	VBR + 0x100	0x1E0
NMI	VBR + 0x600	0x1C0
外部割込み IRL = 0000 IRL = 0001 IRL = 0002 : IRL = 1110	VBR + 0x600	0x200 0x220 0x240 : 0x3C0
内蔵周辺からの割り込み	VBR + 0x600	0x400 : 0x760

一方，SH-3以降では例外ベクタテーブルを参照せず，直接共通の例外ベクタ(リセット用と他に3種類)にジャンプする方式に変更された(**表7**)．実際にどの種類の例外が発生したかはEXPEVT(一般例外用，TLBミスも？)，INTEVT(割り込み用)，TRA(TRAPAのパラメータの4倍)レジスタに格納されている例外要因の値で区別する．この方式は，MIPSの方式に近い．具体的には，リセットがP2(非キャッシュ，非TLBマップ)領域の0xA0000000に固定されている．割り込みと一般割り込み例外は，ベクタベースレジスタ(VBR)が示すアドレスからのオフセットとなっている．割り込みはVBR + 0x600，TLBミスがVBR + 0x400，一般例外がVBR + 0x100である．

SHにおける割り込みのアーキテクチャは，SH-1からSH-4へとMPUが進化するにつれて簡略化される方向にあるようだ．

▶割り込み/例外の動作

実際の割り込み処理の流れを示す．(1)と(2)は外部割り込みの場合で，例外の場合は直接(3)に移行する．(7)～(10)がソフトウェアでの処理である．

(1) 割り込み要求が発生(IRL0～IRL3)
(2) SRの割り込みマスク(I0～I3 = IMASK)と比較して優先度が高ければ割り込みが発生する
(3) 例外要因レジスタ(INTEVTなど)に割り込み要因コードがセットされる
(4) SRとPCがSSRとSPCに退避される
(5) SRのブロックビット(BL)，モードビット(MD)，レジスタバンクビット(RB)が1にセットされる

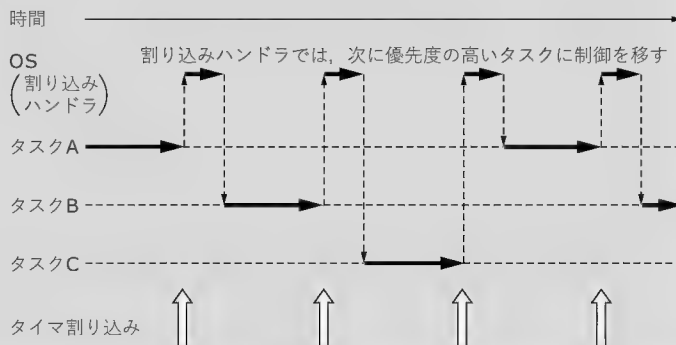
Column 3

割り込みとタスク切り替え

マルチタスクの実現方法として，プリエンプティブ方式というものがある．これはタイマ割り込み(一定間隔で発生する割り込み要求)を契機として，タスクを切り替える方式である．実行中のタスクは，時間がくる(タイマ割り込みが発生する)と割り込みを受け付けて，その実行を強制的に中断し，制御をOSのタスク制御プログラムに移す．OSは中断したタスクのコンテキスト(実行を再現するためのPCやレジスタなどの値)を退避し，次に優先順位の高いタスクのコンテキストを回復して，そのタスクに実行の制御を移す．

このようにして，一つしかないMPUが，複数のプログラム(タスク)を，短い時間に少しずつ実行して行くことで，それらが同時に動作しているように見せる．これがマルチタスクによる並行処理の正体である．そして，マルチタスク動作を行うためのキーポイントとなるのがタイマ割り込みという割り込みの1種なのである．

三つのタスク(タスクA，タスクB，タスクC)が存在する場合の，タスク切り替えの行われるしくみのイメージを**図C**に示す．タイマ割り込み自体はMPU内部のタイマカウンタにOSが値を設定し，そのカウンタの値が一定値に達すると割り込み要求が発生する．また，タイマ割り込みが発生するごとにOS内でタイマカウンタは設定し直される．

〔図C〕プリエンプティブなタスク切り替えのイメージ

(6) 割り込みハンドラへジャンプする
(7) 多重割り込みを許可する場合は，
- SSR，SPCの値をスタックに退避する
- IMASKを許可する優先順位に設定する
- BLビットを0にする(割り込み許可)

(8) 割り込み処理を行う(BL＝0の場合はより優先度の高い割り込みを受け付け可能)
(9) 多重割り込みを許可する場合は，
- BLビットを1にする(割り込み禁止)
- スタックからSSR，SPCを回復する

(10) RTE命令を実行する
(11) SSR，SPCがSR，PCにセットされる(割り込まれた元にジャンプする)

なおSH-4では，割り込みコントローラの設定(ICRレジスタのIRLMビット)で，4ビットの優先順位入力を独立した4本の割り込み要求として利用することもできる．この場合，IRL0，IRL1，IRL2，IRL3の優先順位は，それぞれレベル13，10，7，4として扱われる．

● 各MPUの特徴のまとめ

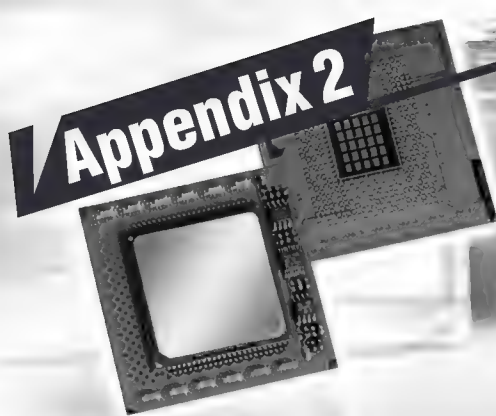
こうして各アーキテクチャの割り込み/例外処理の実装を見ると，次のようなことがわかる．CISC(というか古いMPU)では，ハンドラのアドレスが格納されたテーブルを参照して分岐先を決定するのに対し，RISCでは割り込み/例外の種類に応じた特定のアドレスに直接分岐することである(SH-1/SH-2を除く)．これは，少しでもメモリ参照回数を低減して性能向上を図る，RISCの方針が表れたものかもしれない．

まとめ

割り込みや例外というものは，MPUの動き自体は単純なものである．なぜ，そのような機構が提供されているのか，その思想的背景を理解することのほうが難しい．今回の説明でわかっていただけたであろうか(じつは少し不安)．

割り込みといえば，何かの仕事を中断して別の仕事をするというイメージである．この場合，後で中断した処理を再開するために，スタックに戻りアドレスなどの復帰情報を退避することが前提である．このため，現実の生活では，割り込み仕事が連続して頻発すると，「スタックがオーバフローして，さっきまで何をやっていたかわからないよ」と悲鳴を上げることがしばしばである．これも職業病だろうか．

なかもり・あきら　フリーライタ

Appendix 2

誤り検出/訂正符号やシステムの多重化など

高信頼性をサポートする機能

中森 章

はじめに

コンピュータの応用は，さまざまな分野に広がっている．その中でも，金融機関のオンライン処理，医療機器，ロケットや人工衛星，交通機関制御への応用は，高い信頼性を要求される．これらの分野では，コンピュータが停止すると重大な事故を引き起こしてしまう．しかし，どんなに注意していても故障(フォールト)は発生する．その場合でも，被害を最小限に食い止めるしくみがコンピュータに求められる．このように，故障に強く無停止動作を実現するシステムを「フォールトトレラントシステム」という．

また，大型計算機やEWSなどの一般的なコンピュータでも，ある程度の高信頼性は重要である．いったん故障が発生すれば，修復や保守のコストが高くついてしまう．それを避けるためのしくみは，RAS(信頼性：Reliability，可用性：Availability，サービス性・保守性：Serviceability)として，高性能コンピュータの特徴の一つとなっている．

高信頼性は，MPU，メモリ，記憶装置，I/O装置など，システムのすべての構成要素に要求される．その本質は，故障の検出にある．ここでは，MPUが提供するフォールトトレラントシステムのサポート機能について説明する．

● 誤り検出/訂正符号

MPUに直接されている周辺機器には，メモリとI/Oがある．I/Oに関しては，同じアドレス(ポート)であっても入出力される値は場合によって異なるので，その値が正しいかどうかを判断する方法はない．しかしメモリに関しては，与えられたアドレスに対するメモリの内容は意図的に変更しない限り不変であるはずなので，その値が正しいか否かを判別するのは重要である．メモリに記憶されているデータは放射線やノイズによって破壊されることもあり，MPUがリードする値がいつも正しいとは限らない．高信頼化システムではメモリ内容の正当性を保証する必要がある．

メモリの誤りを検出する方法として通常行われるのが，メモリライト時に，そのデータを加工した特殊な値(シンドロームと呼ばれる)をデータと一緒にメモリに格納しておき，その後のメモリリード時に，メモリからデータと同時にリードしたシンドロームとデータから新たに計算されるシンドロームを比較して一致するか否かを検査する(**図A**)．シンドロームが一致すればそのデータは正しいとみなせる．この際，データからシンドロームを再計算する機構と比較する機構はMPU内部に備わっている．誤りを検出した場合は例外を発生する．

このような誤り検出符号では，パリティとECC(Error Checking and Correcting)が有名である．ただ，パリティにしろECCにしろ，シンドロームを計算する機能が高速動作時においてクリティカルパスとなるため，組み込み用途などの安価なMPUでは採用されない．

▶もっとも単純なパリティ

誤り検出符号でもっとも単純なものはパリティである．これは，データ内の全ビットの排他的論理和を取った1ビットの値である．パリティを含めて結果が0となるもの(つまり1の数が偶数)を偶数パリティ(even parity)，結果が1となるもの(つまり1の数が奇数)を奇数パリティ(odd parity)と呼ぶ．

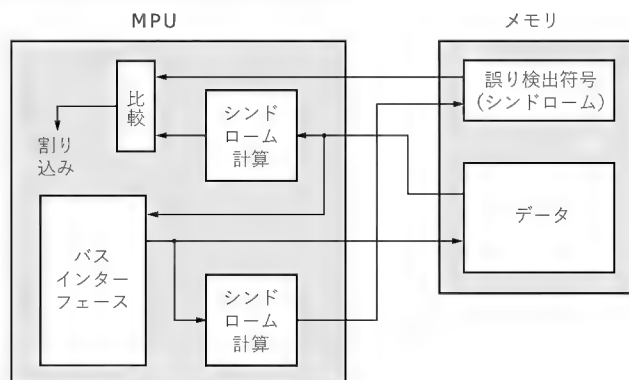
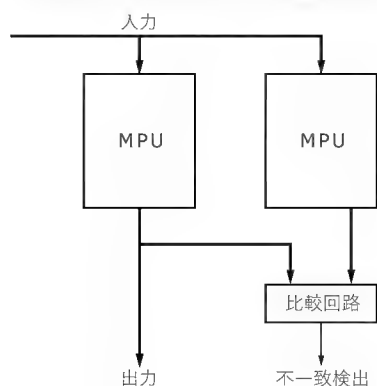
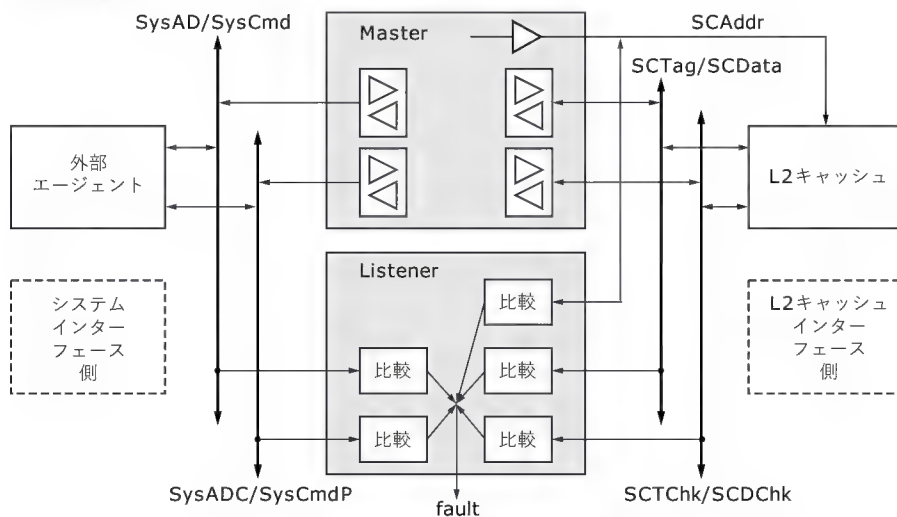
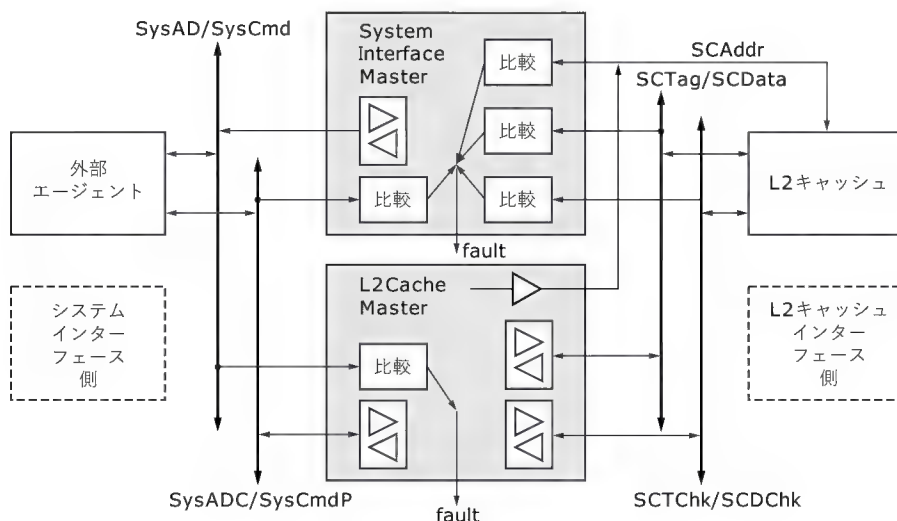
パリティは，その生成原理から，偶数個のビットが誤った場合でも正しいデータとみなしてしまう．その危険を少しでも低減するため，データをいくつかに分割して，それぞれをパリティで管理する．たとえば，32ビットデータであれば4分割して，8ビットずつに対する計4ビットパリティを用いる．

▶パリティより複雑なECC

ECCはパリティよりも複雑な符号化を用いる．パリティと異なり，データに誤りがあった場合，それを訂正することができるのが特徴である．何ビットの誤りを訂正できるかにより，いろいろな符号化方法があるが，実現のしやすさとハードウェア規模を考慮してSEC-DED(Single-bit Error Correcting and Double-bit Error Detecting)コードが多用される．これは，その名のとおり，1ビットまでの誤りを訂正し，2ビットまでの誤りを検出することができるコードである．

SEC-DEDのシンドロームの計算方法は複雑なので，ここでは言及しない．簡単にいうと，データの各ビットを数種類の係数列と積和することで数ビットのシンドロームを得ることができる．何種類の係数列が必要かはデータのビット長に依存する．たとえば，64ビットデータに対しては8系列が必要である．結果として，64ビットデータからは8ビットのシンドローム(ECCコード)が生成される．

メモリに保存されているECCコードと計算したECCコードの排他的論理和を取った場合，結果が0であればデータは正しいとみなせる．結果が0でない場合は，それをデコードすることでメモリデータ長に等しい値を得ることができる．もし，その値の中に1であるビットが一つだけあれば，メモリデータの対応位置のビットが誤りである．つまり，元のデータと排他的論理和を取れば，データを訂正できる．もし，1であるビットが複数あれば，メモリデータが誤りであ

〔図A〕誤り検出符号を使用するシステム

〔図B〕冗長構成(外部回路で不一致検出)

〔図C〕2重化システムの例

(a) Master-Listener構成

(b) Cross-Coupledチェック構成

ることを表す．この場合は，対応するビット位置が誤りというわけではない．

ECCコードを採用する場合，メモリリードと同時にデータを訂正するためには，厳しいタイミングが要求されるため，動作スピードに影響を与えることになる．そこで，MPUでの実装では，ECCコードの排他的論理和を取った時点で，その値が0でなければ例外を発生し，訂正処理をソフトウェアにまかせることが多い．

● 冗長性による高信頼化

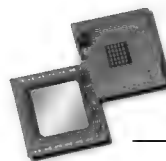
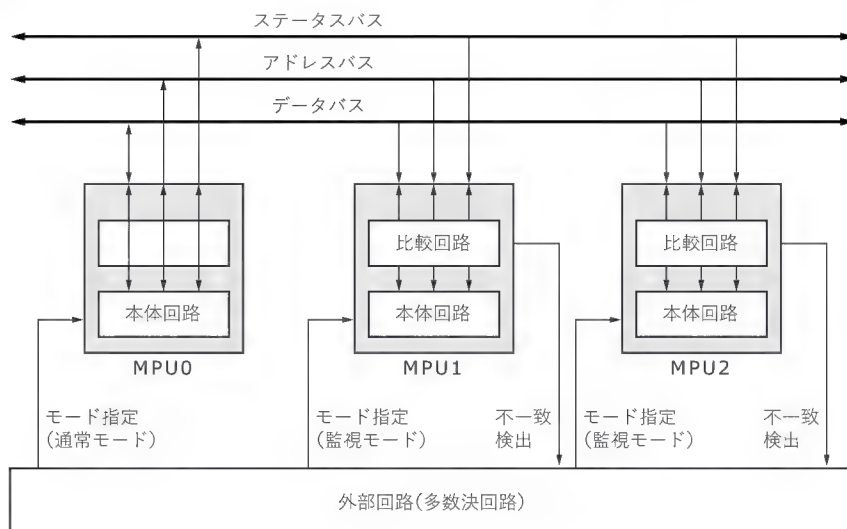
MPUの高信頼化では，共通の入力に対し，複数のMPUを同時実行させて，各MPUの出力を比較し，その一致を検査する．これをロックステップ(lock-step)操作という．**図B**のように外部回路で一致を検査する方式もあるが，回路規模が大きくなるため好まれない．通常はMPU自体が監視モードをもっている．

監視モードでは，MPUの出力端子は入力端子に切り替わり，対応する端子の出力(これが監視モードへの入力)と自身の出力をMPU内部で比較する．もし，不一致が発生すれば，専用端子を活性化して，故障の発生を外部に通知する．

ロックステップ構成における注意点は，複数のMPUを完全に同期化して動作させる必要があるということである．とくに割り込みなどの非同期入力に関してはタイミングがずれないように注意しないと，MPU間の動作がずれてしまうことがある．この場合は，たとえ正常動作していても不一致が生じることがある．このため，MPU間で定期的に同期を合わせる機構をもつMPUもある．たとえば，特定のMPUをストールさせて，待ち合わせを行うための入力信号が提供される．

▶ 2重化システム

2重化システムでは，二つのMPUを並列に接続してロックステップ操作をする．一方が通常モード，片方が監視モードである．

監視モードをもつMPUとしてはMIPSのR4400がある．R4400では2種類の2重化システムをサポートする．一方が通常動作をし，片方が監視動作をするMaster-Listener構成〔**図C(a)**〕と，一方がシス

〔図D〕3重化システムの例

テムインターフェースを駆動しながらL2キャッシュインターフェースを監視し，片方がシステムインターフェースを監視しながらL2キャッシュインターフェースを駆動するCross-Coupledチェック構成〔**図C(b)**〕である．つまり，通常モードを含めて4種類の動作モードをもつことになる．R4400はブート時に4種類の動作モードを指定できる．

▶3重化システム

2重化システムでは，通常モードと監視モードのどちらのMPUが故障したのか知ることができない．そこで，MPUを三つ並列接続して多数決で故障したMPUを特定する構成がある．一つが通常モードで，残りの二つが監視モードである．この構成は，一般に，TMR(Triple Mudular Redundancy)構成(**図D**)として知られている．

この構成では，監視モードの一つのMPUが故障した場合は，構成を2重化システムに変更してある程度動作を継続できる利点がある．その間に，部品の交換や修理を行うことができ，時間稼ぎができる．

宇宙開発事業団(NASDA)が打ち上げを行ったH2Aロケットの姿勢制御，エンジン制御にはNECのV70が使用されている．V70もFRM(Function Redundancy Monitor)と呼ばれるロックステップ動作をサポートする．H2AロケットではV70の3重化システムが使われているそうである．

● 監視タイマ

複数のMPUで冗長構成を採らず，一つのMPUで故障検出を行う方法として監視タイマがある．これはウォッチドッグタイマ(Watchdog Timer)として知られている．

これは単純なタイマである．初期値を設定し，それがカウントアップ，あるいは，カウントダウンされて，一定の値に達すると割り込みを発生する．

プログラムでは，いくつかのチェックポイントで，ウォッチドッグタイマに初期値を設定し直す．プログラムの実行に何か不都合が発生し，ある時間内にウォッチドッグタイマを設定できなければ，タイマが規定値に達し，割り込みが発生して故障を通知するしくみである．故障発生時はMPUをリセットする必要があるので，ウォッチドッグタイマを専用にもつMPUでは割り込みの代わりにリセット例外を発生するものもある．

＊　　＊

フォールトトレラントシステムにおいて，MPUに要求される機能について説明してきた．使い捨ての感がある組み込み機器やPCでは，低コスト化の要求が強いため高い信頼性を提供することは少ないが，こういう世界もあることを知っておいてほしい．

なかもり・あきら　フリーライタ

命令セットアーキテクチャの変遷

中森　章

コンピュータが誕生して以来，さまざまなアーキテクチャのMPUが登場してきた．本章では，「究極のCISC」とも呼ばれるV60/V70から，MIPS，ARM，i860，88000，SPARC，PowerPC，PA-RISC，Alphaの命令セットアーキテクチャについて解説する．豊富な命令数とアドレッシングモードを備え複雑化したCISCの反省からRISCが生まれ，そしてRISCもまた複雑化していくようすがわかるだろう．（編集部）

はじめに

CISC(Complex Instruction Set Computer＝複雑な命令セットのコンピュータ)という言葉は，命令セットが複雑な昔のコンピュータの命令セットを揶揄した，**RISC**(Reduced Instruction Set Computer＝縮小された命令セットのコンピュータ)の研究者が創造した言葉である．CISCとRISCの**命令セット**には，どのような違いがあるのだろう．ここでは具体的なMPUの命令セットについて解説することで，それぞれの特徴を見ていきたい．そして，MPUの進化や変遷につれて命令セットがどう変わっていったのかを知っておこう．それは，とりもなおさずMPU自体の歴史ともいえる．

コンピュータアーキテクチャとは

● IBM System/360の時代

コンピュータアーキテクチャとは何だろうか．コンピュータアーキテクチャという言葉が初めて使われたのは，じつはそれほど古くない．1964年，Gene M.Amdahlらが IBMジャーナルに寄稿した“Architecture of the IBM System/360”という論文の中でである．

そこでの定義は，プログラマから見たコンピュータということ．つまり，命令セットと命令セットの実行モデルということだった．その本質は，アーキテクチャの設計と特定の実装方式を切り離して考えることにある．同じアーキテクチャをもつコンピュータは「ファミリ」と呼ばれ，同じファミリ内であれば，ハードウェアの実装方法やファームウェアが異なっても，プログラムに互換性がある．プログラマは命令セットだけを気にしていればよい．この概念はファミリという考え方を一般的にし，IBM System/360やSystem/370だけでなく，PDP-11やVAX，680x0，x86アーキテクチャの開発に大きな影響を与えた．

● コンピュータの方式を示す大きな概念

しかし技術の発展により，Amdahlらの定義は古くなってきた．プログラムの実行はライブラリ，OS，システム構成に影響され，命令セットが同じであっても互換性があるとは限らない．いまや互換性は，OSとのインターフェースやさまざまな規格を統一しないと実現できない．またアドレス空間のビット数，仮想記憶やキャッシュの構成などの実装方式も，互換性に影響を与えることがある．

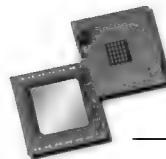
この意味で現在では，アーキテクチャという言葉はコンピュータの方式を示す非常に大きな概念になっている．そのため，特定の方式を言及する場合は，命令セットならば**命令セットアーキテクチャ**，実装方式ならば**マイクロアーキテクチャ**(ハードウェアアーキテクチャ)，システム構成なら**システムアーキテクチャ**などと，固有の名称を使用する．

さて，先月号と今月号の第3章までは，コンピュータのマイクロアーキテクチャをおもに解説してきたが，ここでは命令セットアーキテクチャに注目する．

● 基本的な命令機能はどれも同じ

MPUの命令セットアーキテクチャの基本はどれも同じである．データの移動，データの加減乗除，論理演算(AND，OR，XOR，NOT)，比較と条件分岐命令である．これらは，整数データや浮動小数点データを処理する演算として必ず定義されている．場合によってはNOTがNORであったり，比較と条件分岐が一つの命令になっていたりするが，実現できる操作は同じである．これを基本として，手続き呼び出し命令や，割り込みや動作状態を操作するシステム制御命令が付加される．

MPUの種類によっては，整数と浮動小数点以外のデータ型をサポートするものもある．それら新しいデータ型に対しては専用の演算命令が用意される．たとえば，連続する整数データを文字列またはビット列とみなして，それらに特殊な処理を施す命令(文字列の転送，文字列の比較，文字やビットのサーチ)が考えられる．

また，いくつかの命令機能を一つの命令で実現させるようにすると，命令数はどんどん増加していく．その極端な例がCISCであろう．

● 2オペランド形式と3オペランド形式

MPUの命令は，命令コード(オペコード)とオペランドから成る．このオペランドの個数が，命令セットアーキテクチャを特徴付ける一要素となっている．これには，オペランドを2個もつ**2オペランド形式**と，オペランドを3個もつ**3オペランド形式**がある．

一般的なデータ処理を考える場合，転送はソースオペランドとデスティネーションオペランドの二つが決まれば実現できるが，演算は二つのソースオペランドと一つのデスティネーションオペランドが必要である．つまり，ソース1とソース2を演算して結果をデスティネーションに格納する．よって，オペランドの個数としては3個がもっとも自然である．

しかし，世の中のMPUは2オペランド形式を採用するものも多い．この形式は，演算において，片方のソースをデスティネーションと兼用する．また，この形式は片方のソースが破壊されるという意味で，3オペランド形式よりもプログラミングの自由度が低い．では，なぜ2オペランド形式が採用されるのかというと，それは命令長を節約できるからである．

● 命令長について

たとえば，アーキテクチャ的に32本のレジスタを使用できる場合を考えると，レジスタを指定するためにはオペランドに5ビット分の領域が必要である．一つの命令内でレジスタを指定する総ビット数は，2オペランド形式では$5 \times 2 = 10$ビット，3オペランド形式では$5 \times 3 = 15$ビットである．もし，命令長が固定されていると考えると，3オペランド形式では2オペランド形式に比べて5ビット分，オペコード指定に使えるビット数が減ってしまう．つまり，命令の種類が制限されてしまう．逆に考えると，同じ数の命令を実現するには，3オペランド形式は2オペランド形式よりも命令長が長くなる．

一般的にRISCは，命令デコードのしやすさとの兼ね合いで32ビット固定長の命令を採用する場合が多い．しかし，x86に代表されるCISCは，バス速度が遅かった昔の名残で，命令コードを短時間に取り込む工夫，すなわち，命令長を短くする工夫をしている．その一環が2オペランド形式である．さらにCISCでは，多様なアドレッシングモードを指定可能にするために，ただでさえ長くなりがちな命令長をバイト単位の可変長にすることで対応している．

● アキュムレータ形式/スタック形式

オペランド形式としては，2オペランド形式，3オペランド形式のほかに，演算可能なレジスタをアキュムレータ(特殊レジスタ)に限定する**アキュムレータ形式**，演算をスタック上で行う**スタック形式**がある．

アキュムレータ形式は，演算器に直結するレジスタをアキュムレータに限定する．オペランドの一つがアキュムレータであることがわかっているので，その分，命令コードを短くできる利点がある．これは，トランジスタの集積規模が小さく，すべてのレジスタを演算器に接続できなかった昔のMPU，たとえば，8080やZ80に見られる．

スタック形式は，演算をスタック上でしか行わない点で，アキュムレータ形式の特殊なものとみなすこともできる．しかし，アキュムレータが，通常は一つしかないのに対して，スタックは理論上無限の個数があり，複数の中間結果を同時に格納できるという点で，式の計算を実現するのに便利である．

〔表1〕V60/V70の命令の種類

●転送	●10進演算
●整数演算	●浮動小数点演算
●比較	●手続き呼び出し
●論理演算	●分岐
●シフト/ローテート	●PSW(Program Status Word)操作
●実効アドレス計算	●MMU制御
●単一ビット操作	●入出力
●ビットフィールド(挿入，抽出，比較)	●タスク制御
●ビットストリング転送	●アトミック(不可分)命令
●文字ストリング転送	

2 CISCの命令セット

● 複雑な命令セットのコンピュータ＝CISC

RISC誕生以前は，いわゆるCISCしかないわけで，CISC命令セットがコンピュータの命令セットの原点である．

CISCの命令セットは，一部に簡単な処理を行う命令もあるが，大半は複雑な処理を行う命令の集合である．複雑とは，一つの命令で多くの処理を行うことである．これはメモリのアクセス時間が遅かった時代にコンピュータの実行性能を高めるための自然な選択である．少ない命令でプログラミング言語のコンパイラやインタプリタで行う処理を効率的に実現したり，OSの操作を効率的に実現したりするための工夫が盛り込まれている．

その特徴をV60/V70(NEC)で見ていこう．このMPUの命令セットが完成した時期はCISCの後期に属し，その意味で，ほかのCISCの命令セットの「いいとこ取り」であり，究極のCISCともいえるからだ．

● V60/V70の命令セットの特徴

V60/V70の命令の種類を**表1**に示す．命令長は1バイトから22バイトまで存在し，2オペランド方式である．そして，ソースとデスティネーションは21種のアドレッシングモードを独立に指定できる．これをV60/V70では「対称性」と呼んでいる．なお，命令長を短縮するために，片方のオペランドがレジスタの場合は短縮型の命令形式が用意されている．また，データ型は次に示す14種で，それぞれのデータ型に関してすべての演算(意味がある場合)が定義されている．これを「直交性」と呼んでいる．

- 整数(バイト，ハーフワード，ワード，ダブルワード)
- ポインタ
- ビット
- ビットフィールド

- ビットストリング
- 10進数(パック，アンパック)
- 文字ストリング(バイト，ハーフワード)
- 浮動小数点データ(単精度，倍精度)

このように，V60/V70の命令セットは対称性，直交性に優れていることを特徴としている．これはプログラムの書きやすさはもちろんだが，コンパイラの作成を容易にする意図がある．

● 命令の特徴

V60/V70で特色のある命令を見ていこう．その項目を検証することで，CISCの命令セットがどのような項目を重要と考えていたかが推測できる．

▶続き呼び出し関連

手続き呼び出し命令は，高級言語のコンパイラを実現するための要である．Cコンパイラにおいて，手続き呼び出しは一般に次のようなシーケンスを取る．

- 引き数をスタックに積む(PUSH命令)
- 手続きを呼び出す(CALL命令)
- ローカル変数のためのスタックフレームを作成する(PREPARE命令)
- レジスタ変数に使用するレジスタを一括して退避する(PUSHM命令)
- 手続きの実行
- 退避したレジスタを一括して回復する(POPM命令)
- スタックフレームを解放する(DISPOSE命令)
- 手続きから復帰する(RET命令)
- 引き数領域を開放する(POP命令またはADDによるスタックの補正)

このように，V60/V70ではそれぞれの処理に対応する専用命令が用意されている．V60/V70の手続き呼び出しで特徴的なのは，アーギュメント(引き数)ポインタという概念である．これは引き数を参照するためのベースレジスタであり，CALL命令によって値が設定される．Cコンパイラでは，アーギュメントポインタは，CALL命令実行時のスタックのトップだが，FORTRANやCOBOLでは別の場所になる．それらに対処したわけである．ローカル変数に関しては，他のアーキテクチャと同じくフレームポインタをベースとして参照する．またRET命令は，オペランドの値でスタックポインタを補正することもできる．つまり，スタックにある引き数領域をRET命令実行時に解放することもできる．Cコンパイラでは呼び出し側で引き数領域を解放するので，これはPASCALコンパイラ用である．

なお，これらの手続き呼び出しシーケンスは，VAXのそれに非常に強い影響を受けていることを付け加えておく．

▶ビットストリング操作/ビットフィールド操作

この命令はビットマップグラフィックのデータ処理に用いる．この命令により，メモリ中の任意のビット位置から任意のビット長のビット列同士にNOT，AND，OR，XOR，AND-not，OR-not，XOR-notなどの論理演算を施すBitBlt処理を行える．

ビットストリング操作のうち，ビット列の連続する0または1を計数する命令は，現在でもFAX処理や画像の圧縮伸張に応用できる．

ビットストリング命令に似た命令でビットフィールド命令もある．これはメモリの任意の位置から指定したビット長のフィールドを抽出/比較したり，メモリの任意の位置から指定したビット長のフィールドにデータを挿入したりできる．この命令も画像の圧縮伸張に使える．

▶文字ストリング操作

C言語でいうところのstrcpy，strcmp，strlenなどのライブラリ関数の機能を1命令で実行できる．転送の単位は8ビットと16ビットがあり，漢字コードの転送も考慮している．ソースとデスティネーションの文字ストリングがオーバラップする場合も正常な転送ができるように，逆方向から転送する命令もある．これらの命令は大型計算機のACOSのデータ転送命令を参考にしたといわれている．

▶10進演算

COBOLなどでの使用を考慮し，10進数の文字コードを直接加減算する命令がある．BCD形式(パック型)の10進数も演算できる．現在はどうか知らないが，以前，世界でもっとも多く使われているコンピュータ言語はCOBOLだった．10進演算命令は，COBOLにおける数値処理を高速に処理するためのものである．

▶MMU制御

ATE(Area Table Entry)やPTE(Page Table Entry)といったアドレス変換テーブルの内容を，その各エントリに関連する仮想アドレスによる指定で直接リード/ライト可能なUPDATE，GETATE，UPDPTE，GETPTE命令や，仮想アドレスと対になる物理アドレスを得るGETRA命令などがある．また，各実行レベルからのアクセスの可否を判断するCHKAR/CHKAW/CHKAE命令，実行レベルを変更するCHLVL命令がある．TLBの操作に関しては，指定した仮想アドレスにヒットするエントリを無効化するCLRTLB命令とすべてのエントリを無効化するCLRTLBA命令がある．TLB内容の入れ替えは自動的に行われるため，TLBの内容を直接操作する命令はない．

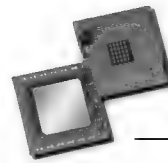
▶コンテキスト切り替え

マルチタスク環境下でのタスク切り替えを1命令で実行するコンテキスト切り替え命令(LDTASK/STTASK)がある．この命令は，V60/V70のレジスタセットや仮想記憶情報を選択的にメモリ中にあるタスク制御ブロックの内容と入れ替えることができる．

▶アトミック命令

マルチプロセッサ環境でのセマフォを実現するための，テストアンドセット命令(TASI)とコンペアアンドスワップ命令(CAXI)がある．これらの命令はバスをロックして操作を行うアトミック命令である．

▶非同期トラップ

これは命令ではないが，OSの機能をサポートするしくみで

ある．非同期とは，トラップが発生する条件があってもただちに例外処理に移行するのではなく，あと（RETI 命令の実行時）まで遅延させることを意味する．つまり，条件の成立と発生が同時でないことを示す．V60/V70では，OSのための非同期システムトラップとユーザータスクで使用できる非同期タスクトラップが提供される．

● V80での高速化項目

V60/V70の後継機種であるV80の命令セットは，基本的にV60/V70と同一である．機能的にはV60で完成していたと考えられたからだ．V60/V70の命令セットは，コンパイラの作成しやすさ，OSの書きやすさを第一に考えているのがわかる．V80ではこれをさらに高速化することに注力している．具体的には，次に示すような項目である．

- 基本命令のハードワイヤードロジック化
- SP（Stack Pointer）のフォワーディング
- CALL/RETの高速化，RETIの高速化
- 文字ストリング操作のハードウェア化
- ビットフィールド操作の高速化
- ビットストリング操作の高速化
- TLB入れ替えのハードウェア化
- FPUの高速化，乗算器
- 分岐予測機能の採用
- キャッシュの採用
- アトミック命令の追加（ADDI/SUBI/ANDI/ORI）

これらの機能の導入により，**表2**に示すような性能向上が得られたという．現在のMPUの実行クロック数からみればかなり低性能であるが，当時としてはかなり高速な部類に入る．

3 崩れた神話 —— RISCへ至る道

● 直交性に優れた命令を用意したが……

コンパイラに優しいCISCの命令セットは，良質のコンパイラの登場を約束するはずだった．しかし，現実はその思惑どおりには進まなかった．多種多様な命令とアドレッシングモードがあればコンパイラを作りやすいのは確かである．しかし，コンパイラが生成する命令コードの性能という観点で見ると，パイプラインを有効活用できない複雑な命令は不利ということがわかってきた．

CISCの時代にはまだバス速度が遅く，複雑な1命令がいいか，単純な複数命令がいいかということは一概にはいえなかった．しかし，内蔵キャッシュが一般的になって，初めて単純な複数命令のほうが有利となった．また，パイプライン処理を前提とすると，一つの命令で多くの処理をするより，単純な命令に分解して実行するほうがスループットが高い．さらに，コンパイラの最適化技術が進むと，ロード/ストア以外ではメモリ参照を行わなくなる．つまり，命令のオペランドはレジスタだけで事足りる．そして，パイプラインを乱すメモリ間接アドレッシングはほとんど使用しなくなった．

〔表2〕V80での高速化の実際（実行クロック数）

機　能	命令，条件など	V70	V80
転送	MOV.W　reg,mem	4	2
	MOV.W　mem,reg	4	2
整数演算	ADD.W　reg,mem	2	2
	ADD.W　mem,reg	4	2
	ADD.W　mem,mem	8	4
整数乗算	MUL.W	23	9
整数除算	DIV.W	43	39
シフト	SHA.W	17	3
分岐	Branch Taken	11	2
	Not Taken	4	4
手続き関連	CALL & RET	44	21
ビットフィールド	EXTBFZ	30	10
	INSBFZ	28	10
文字列操作	MOVCU.B(n bytes)	30＋5n	19＋1.25n
浮動小数点（単精度）	ADDF.S	120	36
	MULF.S	116	44
	DIVF.S	137	75
浮動小数点（倍精度）	ADDF.L	78	75
	MULF.L	270	110
	DIVF.L	590	553
割り込み復帰	RETIS	80	22
コンテキスト切り替え	LDTASK(44words)	347	157
	STTASK(44words)	200	121
TLBミス処理	異なるエリア	58	11
	同一エリア	58	6
割り込み応答	ハンドラ実行まで	165	27

● 単純命令を高速に実行

CISC命令の高速化の過程で生まれてきた考え方の一つは，使用頻度の高い命令を高速に実行することである．性能にクリティカルな命令を高速に実行するためにワイヤードロジック化したり，専用ハードを導入することが自然と考えられた．しかし，複数の処理を1度に実行する命令を高速に実現するためには大規模な専用ハードが必要になり，チップ面積も必要になってしまう．

この状況に新たな道を見出したのがRISCの研究である．統計をとると，単純な命令ほど使用頻度が高く，かつ性能に効いてくる．単純な命令を高速化するためには大規模なハードウェアは不要であり，その少し単純なハードウェアが全体の性能に効く．単純な構造のため動作周波数も上げやすい．

4 誕生初期のRISC

● ロード/ストアアーキテクチャと単純なアドレシングモード

初期のRISCとして有名なのはIBM801，バークレーRISC I，スタンフォードMIPSである．これらの命令セットの特徴を以下に説明するが，その前に，いわゆるCISCとこれらの命令の特徴を**表3**に示しておく．一見すると，「縮小命令セットコン

ピュータ」の名のとおり，RISCの命令数はCISCの命令数よりもかなり少ない．しかし，CISCでは同じ種類の命令でも処理するデータサイズによって異なる命令とみなされるため，命令数が多く見える．

一方RISCでは，ロード/ストア以外は，レジスタ間で演算がなされるため，演算に関してデータサイズという概念がない．後述の命令セットの具体例を見ると，CISCとRISCの命令数(種類)は大差ないことがわかる．RISCをRISCたらしめている特徴は，ロード/ストアアーキテクチャと単純なアドレッシングモードであるといえる(**表4**)．

● パーシャル(部分)レジスタライト

CISCからRISCへの以降の間に多くのユーザーが忘れ去っている特徴に，**パーシャルレジスタライト**がある．これは，たとえばレジスタが32ビット長の場合，8ビットまたは16ビットの演算に対して，それぞれのビット長に対応する部分だけしか変更されないというものである．つまり，8ビット演算なら，レジスタの上位のビット31～8は変更されない．

これはx86のHIレジスタ，LOレジスタへの独立アクセスあたりにルーツがあるように思う．x86と同程度に古いアーキテクチャであるMC68000も，同様の特徴を有していた．

このパーシャルレジスタライトの概念をくつがえしたのがRISCアーキテクチャである．演算自体はデータ長というものはなく，必ずレジスタ全体が変更される．データ長という概念をもっているのはロード/ストア命令のみである．

ロード命令に関しては，レジスタ長にゼロ拡張/符号拡張されて格納される．つまりレジスタ全体が変更される．ストア命令に関しては，メモリに対して部分ライトされる．これはCISCと同様である．

現在のMPUではx86以外にはパーシャルレジスタライトの特徴は見受けられない．x86でも，パイプラインがストールするので，パーシャルレジスタライトの使用は推奨されていない．

● 条件フラグと条件分岐

CISCのユーザーがRISCのアーキテクチャを最初に見て奇異に思うのは，条件フラグが存在しないということだろう．CISCでは，ほとんどすべての命令で条件フラグが変化する．そして，条件分岐は最終的な条件フラグを参照して分岐するか否かを決定する．

一方RISCには，原則として条件フラグがない．条件分岐はレジスタの値が「0であるか」，「正であるか」，「負であるか」，あるいは二つのレジスタの値が「等しいか」，「等しくないか」という簡単なテストで分岐するか否かを決定する．

RISCで条件フラグをなくしたのは，「フラグハザード」というパイプラインハザードをなくしてパイプライン処理をスムーズに行うためであろう．フラグハザードとは，条件フラグが確定するまで条件分岐命令の分岐先フェッチができずに，パイプラインが停止する状況を指す．

CISCにおいて，条件分岐が参照する条件フラグは，原則的に条件分岐命令の直前の命令で確定する．その命令が加減算のように1クロックで実行できるものなら，それほど害はない．しかし，乗除算命令のように演算に数クロックを要する場合は，その分だけパイプラインがストールする．また，条件分岐命令の前方にある命令列は，条件フラグの値が変わってしまうので，気楽に並び替えることはできない．

RISCにおいて，命令の並び替えは，レジスタの依存性をなくすために日常茶飯事に行われる．この目的のためには条件フラグは邪魔になる．単にレジスタの値を参照するのであれば，レジスタの値はフォワーディングされることもあり，レジスタの値が確定するまでの間のストールを最小限に抑えることができる．もし，RISCで条件フラグを採用するとすれば，その値を予測しフォワーディングすることが必要になり，ハードウェア量の増加をまねく．このため，RISCでは条件フラグを用いないことが多い．

● バークレーRISC I/RISC II

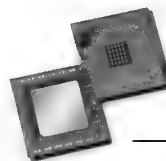
カリフォルニア大学バークレー校のRISCの研究は，高級言語コンパイラが複雑な命令を有効に使えないことに着目することから始まった．プログラム実行時の命令の出現頻度，アドレッシングモード，変数の使われ方などの統計を採り，新しい命令セット設計の指針とした．この研究結果は，1980年に，同大学のPattersonとDitzelによって初めてのRISCに関する論文『The Case for the Reduced Instruction Set Computer』として発表された．

この論文はシングルチップコンピュータに最適なアーキテクチャはRISCであると主張し，次のような利点があると指摘した．

〔表3〕CISCとRISCの命令セットの比較

	CISC			RISC		
	IBM370	VAX11/780	V60	IBM801	RISC I	MIPS
発表時期	1973	1978	1986	1980	1981	1983
命令数	208	303	273	120	23	55
マイクロコード量	54K	61K	23K?	0	0	0
命令長(バイト)	2-6	2-57	1-22	4	4	4
演算対象	reg-reg reg-mem mem-mem	reg-reg reg-mem mem-mem	reg-reg reg-mem mem-mem	reg-reg	reg-reg	reg-reg

〔表4〕RISCの特徴

- 命令の1サイクル実行
- メモリインターフェースは単純なロード/ストアのみ
- レジスタ間での演算
- 単純な形式の固定長命令
- 単純なアドレッシングモード
- 多数の汎用レジスタまたはレジスタウィンドウ
- 遅延分岐
- キャッシュ
- 高級言語コンパイラへの依存

〔表5〕RISC IIの命令セット

整数算術命令	
ADD	加算
SUB	減算
SUBI	減算(逆方向)
S	シフト
論理演算命令	
AND	論理積
OR	論理和
XOR	排他的論理和
ロード/ストア命令	
LDR	ロード
LDX	インデクス付きロード
STR	ストア
STX	インデクス付きストア
フロー制御命令	
JMPX	条件ジャンプ(インデクス付き)
JMPR	条件ジャンプ
CALLX	条件コール(インデクス付き)
CALL	条件コール
RET	条件リターン
RETI	割り込みからの復帰
CALLI	割り込みハンドラをコール
特殊命令	
LDHI	レジスタの上位に値を設定
GETLPC	PCを得る
GETPSW	PSWを得る
PUTPSW	PSWを変更する

〔表6〕MIPSのユーザーレベルの命令セット

整数算術命令	
ADD	加算
SUB	減算
SUBR	減算(逆方向)
IC	バイト挿入
XC	バイト抽出
RLC	レジスタ対のローテート
ROL	左ローテート
S	シフト
MEMSETUP	乗算準備
MSTEP	乗算の1ステップ(2ビット単位)
UMEND	符号なし乗算終了
DSTEP	除算の1ステップ
SET	条件のテスト結果をセット
論理演算命令	
AND	論理積
OR	論理和
XOR	排他的論理和
ロード/ストア命令	
LD	ロード
ST	ストア
MOV	即値またはレジスタの転送
フロー制御命令	
BRA	分岐
JMP	ジャンプ
TRAP	トラップ

▶**チップサイズの縮小**

単純なプロセッサは，少ないトランジスタ数で設計できる．このためCISCに比べて相対的に多くの機能を集積できる．さらに空いた面積を使って，キャッシュやMMU，FPUなどを1チップに内蔵できる．

▶**開発期間の短縮**

単純なプロセッサは，設計にかける労力やコストが少なくて済む．

▶**高性能化**

単純な論理故に高い動作周波数で実行できる．CISCと比べると同じ動作周波数でもIPC(1クロックで実行する平均命令数)が高いので相対的に高性能である．

この論文の理論をバークレー校の大学院生が実践したのが，RISC IとRISC IIである．RISC IIの命令セットを**表5**に示す．これらは当時のCISCよりも単純で，設計の労力も少なかったが，CISCに匹敵する性能を発揮していた．

かくしてバークレーRISCは，後のARMやSPARCアーキテクチャの基礎となるのである．またRISCという言葉は，バークレー校によって初めて使用された．

● スタンフォードMIPS

バークレーRISCと同時期，スタンフォード大学でもHennessyを中心にRISCの研究が行われていた．それがMIPSである．MIPSではハードウェアを簡単にするために，メモリアクセスはワードアクセスのみとし，バイト単位の操作が必要な場合は専用命令を使ってレジスタ上で処理する．また，汎用レジスタは16本だった．

MIPSは2レベルの命令セットをもつ．一つはユーザーレベルの命令で，これはより通常(CISC)に近い抽象的な命令である(**表6**)．この命令セットではレジスタの依存関係を考慮する必要はない．もう一つはマシンレベルの命令で，ALUピース，ロード/ストアピース，制御フローピース，特殊命令(手続き呼び出し)といった部分的な命令コードからなり，リオーガナイザはこれらのピースを組み合わせて実行可能な命令を生成する．このとき，レジスタの依存関係が考慮され，インターロックしなくていいように命令の入れ替えをする．

最初のMIPSプロセッサは実用的といえるものではなかった．しかし，スタンフォード大学の研究者たちは，その研究を推し進め，2Kバイトの内蔵命令キャッシュと256Kバイトの外付けユニファイドキャッシュインターフェース，32本の汎用レジスタ，乗除算用の特殊レジスタ，ゼロレジスタ，5段パイプラインを特徴とする，MIPS-Xというプロセッサを設計した．

● 乗除算命令の処理

MIPSでは，ほとんどすべての命令を1クロックで処理することを目標としている．しかし当然例外もある．浮動小数点演

算と一部のシステム制御命令を除けば，乗除算命令がそれにあたる．乗除算命令は，一般には，1クロックで処理できない．これを通常のパイプラインに組み込むと，パイプラインが乱れて性能低下につながる．

これを回避するため，MIPSでは乗除算を通常のパイプラインとは切り離し，他の演算と並列に処理するようになっている．このため，乗除算の出力(デスティネーションオペランド)として汎用レジスタとは別の専用レジスタを用意している．こうすることで汎用レジスタとの依存性をなくす．その専用レジスタが，HIレジスタとLOレジスタである．

32ビット×32ビットの乗算では積は64ビットであり，上位32ビットがHIレジスタに，下位32ビットがLOレジスタに格納される．一方，32ビット÷32ビットの除算では32ビットの商がLOレジスタに，32ビットの剰余がHIレジスタに格納される．プログラムでは，乗除算命令の後，数命令後に(乗除算の計算が終了したのを待って)，HIレジスタまたはLOレジスタから結果を汎用レジスタに転送することになる．こうすることにより，パイプライン処理に乱れを生じさせない．

● 非整列データ転送命令

MIPS命令セットには非常に特徴的な命令がある．それが**非整列データ転送命令**である．これは，メモリ内の非整列ワード(misaligned words)データを処理する．CISCでは普通にサポートされている機能であるが，たいていのRISCでは非整列なアドレスに対するワードアクセスは例外事象としてトラップを発生する．もちろん，バイト単位でデータを処理すれば，アドレスが整列されていようが整列されていまいが関係ない．しかし，複数のバイトをひとまとめに転送したほうが処理速度が上がる．

MIPSはRISCでありながら，ワードに整列されていないアドレスに対するロード/ストアをサポートする．これはMIPS社の特許であり，かつて互換メーカーのLexra社と訴訟になっていた(現在は和解)のは，この機能の無断使用に関してである．具体的には，次の8命令が用意されている．

(1) LWL(Load Word Left)

ワード内の有効データをレジスタに左詰めする．ロードしたデータでレジスタを部分的に変更する．

(2) LWR(Load Word Right)

ワード内の有効データをレジスタに右詰めする．ロードしたデータでレジスタを部分的に変更する．

(3) SWL(Store Word Left)

レジスタ内に左詰めされたデータをワード内の有効領域にストアする．

(4) SWR(Store Word Right)

レジスタ内に右詰めされたデータをワード内の有効領域にストアする．

(5) LDL(Load DoubleWord Left)

ダブルワード内の有効データをレジスタに左詰めする．ロードしたデータでレジスタを部分的に変更する．

(6) LDR(Load DoubleWord Right)

ダブルワード内の有効データをレジスタに右詰めする．ロードしたデータでレジスタを部分的に変更する．

(7) SDL(Store DoubleWord Left)

レジスタ内に左詰めされたデータをダブルワード内の有効領域にストアする．

(8) SDR(Store DoubleWord Right)

レジスタ内に右詰めされたデータをダブルワード内の有効領域にストアする．

これらの命令を利用すれば，たとえば，R5(転送元アドレス)からR4(転送先アドレス)へのデータ転送をワード単位で行うためには，

```
loop: /* 終了条件は省略 */
  lwr   r8,0(r5)
  lwl   r8,3(r5)
  addiu r5, r5,4
  swr   r8,0(r4)
  swl   r8,3(r4)
  addiu r4, r4,4
  b     loop
```

のように記述できる(リトルエンディアンの場合)．R4とR5の値がワードに整列されている必要はない．

● ARM

ARMはバークレーRISCから，ロード/ストアアーキテクチャ，32ビット固定長の命令，3アドレス形式など，多くの特徴を採用した．しかし次の特徴は採用しなかった．

▶**レジスタウィンドウ**

レジスタの占める面積が多いためコスト面で不採用になったが，その概念は割り込み時のシャドウレジスタに受け継がれている．

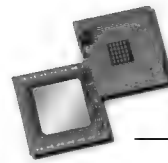
▶**遅延分岐**

例外処理の実装を複雑にするため．

▶**全命令の1クロック実行**

ロード/ストアを1クロックで実行するためには命令とデータを格納するメモリが分離されている必要があり，ARMが対象とするアプリケーションには高価すぎるため．

ARMは命令セットの使いやすさよりも，ハードウェアの実装を簡単に行えることを目標としている．この意味で，ARMの命令セットは，RISCの指針を受け入れながらも，保守的(CISC的)であるといえる．これは単純なハードウェア構成でありながら命令のコード効率を引き上げようとしたためである．**表7**にARM(ARM2相当)の命令セットを示す．

ARM命令の特徴は，すべての命令で条件コードを設定できること，第2オペランドをシフトして演算できること，演算を条件実行できることである．これらの操作をうまく組み合わせれば最小限の命令数で目的の処理を達成することができる．しかし，条件コードがあるため，レジスタの依存性だけに注目し

て命令を並び替えると動作が異なる場合があり，最適化コンパイラ泣かせである．

5 過渡期のRISC

1989年当時，RISCというふれこみで市場に出ていたアーキテクチャの代表は，i860(Intel)，88000(Motorola)，SPARC(Sun Microsystems)である．これらは**表4**に示すRISCの特徴を満たしていた．これらの特徴に加え，i860はグラフィックとベクタ処理の命令を，88000はビットフィールド命令を，SPARCはタグ付きデータ命令というCISC系の命令を有していた．このあたりに過渡期のアーキテクチャという性質を垣間見ることができる．

● i860

i860は，x86とは異なる新しいアーキテクチャを提供する目的で開発された．i386 + 80387の性能を上回る高性能を実現することができ，従来のスーパーコンピュータやミニコンが提供していた科学技術計算や各種のシミュレーションをより小型で安価なシステムで実現できた．

i860は，現在でもDSPの代用品やRAID用のプロセッサとして生き残っている．

● 88000

88000とは，CPUであるMC88100とキャッシュとMMUを内蔵するMC88200というチップの総称である．コードユニット，データユニット，整数ユニット，FPU(加減乗除と変換用の二つ)の計五つのユニットが各自パイプラインで並行動作するという意味でスーパースカラのはしりである．

MC88100は比較結果を反映させる条件コードレジスタをもっていない．比較命令は，ほかの演算命令と同じく，3オペランド命令で，比較結果をデスティネーションレジスタに格納する．条件分岐命令はこのレジスタの値に基づいて分岐する．この構成により，比較命令と条件分岐命令間の命令を自由にスケジューリング(入れ替え)できる．条件コードを使用しないこの方式は，MIPSをはじめとする多くのRISCで採用されている．

● SPARC

SPARCの仕様はオープンアーキテクチャとして，SPARCインタナショナル社によって管理されている．SPARCにはいくつかのバージョンがあり，最新バージョンは9である．バージョン9は64ビットアーキテクチャであるが，(少し前の)典型的なSPARCチップは32ビットアーキテクチャのバージョン7または8に基づいている．

SPARCの最大の特徴は，レジスタウィンドウである．整数ユニットは32ビットの汎用レジスタを136個もっている．このうち8個はグローバルに参照できるが，残りは手続きごとに割り当てられ，引き数の授受を高速に行う．これがレジスタウィンドウで，一つのウィンドウは24個のレジスタからなる．内訳は，R24～R31が手続きの呼び出し元とオーバラップする(引き数用)．R16～R23は手続き内でローカルに使用できる，R8～R15は手続きが呼び出す手続きとオーバラップする．手続きの最初でレジスタウィンドウを切り替えることで，レジスタの値を退避することなく，レジスタを自由に使用できる．SPARCにおける手続き呼び出しのシーケンスは，次のようになる．

- R24～R31に引き数をセットする
- CALL命令を実行する
- 呼び出された手続きはSAVE命令でレジスタウィンドウを切り替える
- 手続きを実行する

〔表7〕ARMの命令セット

データ処理命令	
ADD	加算
ADC	キャリ付き加算
SUB	減算
SBC	キャリ付き減算
RSB	減算(逆方向)
RSC	キャリ付き減算(逆方向)
AND	論理積
ORR	論理和
EOR	排他的論理和
BIC	ビットクリア
MOV	転送
MVN	ビット反転して転送
CMP	比較
CMN	否定して比較
TST	ビットテスト
TEQ	一致テスト
MUL	乗算
MLA	積和
データ転送命令	
LDR	ロード
STR	ストア
LDMIA	多重レジスタロード
LDMIB	多重レジスタロード
LDMEA	多重レジスタロード
LDMED	多重レジスタロード
LDMDA	多重レジスタロード
LDMDB	多重レジスタロード
LDMFA	多重レジスタロード
LDMFD	多重レジスタロード
STMIA	多重レジスタストア
STMIB	多重レジスタストア
STMEA	多重レジスタストア
STMED	多重レジスタストア
STMDA	多重レジスタストア
STMDB	多重レジスタストア
STMFA	多重レジスタストア
STMFD	多重レジスタストア
フロー制御命令	
Bcc	条件分岐
BL	分岐とリンク(サブルーチンコール)
SWI	ソフトウェア割り込み

〔表9〕MC88100の命令セット

整数算術命令	
ADD	加算
ADDU	符号なし加算
CMP	比較
DIV	除算
DIVU	符号なし除算
MUL	乗算
SUB	減算
SUBU	符号なし減算
浮動小数点算術命令	
FADD	浮動小数点加算
FCMP	浮動小数点比較
FDIV	浮動小数点除算
FLDCR	浮動小数点レジスタからのロード
FLT	整数→浮動小数点変換
FMUL	浮動小数点乗算
FSTCR	浮動小数点レジスタからのストア
FSUB	浮動小数点減算
FXCR	浮動小数点制御レジスタとの交換
INT	浮動小数点→整数変換
NINT	Nearest方向の整数変換
TRUNC	Zero方向の整数変換
論理演算命令	
AND	論理積
MASK	論理マスク即値
OR	論理和
XOR	排他的論理和
ビットフィールド操作命令	
CLR	ビットフィールドのクリア
EXT	ビットフィールドの抽出(符号拡張)
EXTU	ビットフィールドの抽出(ゼロ拡張)
FF0	0であるビットの検索
FF1	1であるビットの検索
MAK	ビットフィールドの生成
ROT	レジスタのローテート
SET	ビットフィールドのセット
メモリアクセス命令	
LD	ロード
LDA	アドレスのロード
LDCR	制御レジスタからのロード
ST	ストア
STCR	制御レジスタへのストア
XCR	制御レジスタとの交換
XMEM	レジスタとメモリの交換
フロー制御命令	
BB0	ビットクリア時に分岐
BB1	ビットセット時に分岐
BCND	条件分岐
BR	無条件分岐
BSR	サブルーチンへの分岐
JMP	ジャンプ
JSR	サブルーチンへのジャンプ
RTE	例外からの復帰
TB0	ビットクリア時にトラップ
TB1	ビットセット時にトラップ
TBND	境界チェック時のトラップ
TCND	条件トラップ

〔表8〕i860の命令セット

ロード/ストア	6種
浮動小数点-整数レジスタ間転送	2種
整数算術演算	4種
シフト	4種
論理演算	8種
分岐・コール・トラップ	13種
浮動小数点乗算	6種
浮動小数点加算	12種
デュアルオペレーション	4種
長整数加減算	4種
グラフィックス	10種
I/O	3種
システム制御	6種

- RESTORE命令で元のレジスタウィンドウを回復する
- RET命令(JMPL命令の特殊形)で復帰する(実際はRET命令の遅延スロットにRESTORE命令を置く)

レジスタウィンドウに関しては，多くの利点があることがわかっている．一つ目は手続き呼び出しごとにレジスタの値の退避/回復を行う必要がない点である．二つ目は，高度なレジスタ割り付けを要求しないのでコンパイラがそれほど複雑にならない点である．

● i860/88000/SPARCアーキテクチャの比較

i860の命令セットに関しては，手元に詳細な資料がないので概要のみを**表8**に示す．また，88000とSPARCの命令セットを**表9**と**表10**に示す．

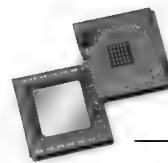
▶特殊命令

標準的な命令セットに加え，i860はグラフィック処理の命令，整数と浮動小数点演算の並列実行(VLIWの特色)，FPUをサポートする．グラフィック処理にはZバッファ操作，Phongシェーディング，ピクセル間演算がある．これらは陰面消去と3D投影に効果的である．しかし，これらの特徴はグラフィック処理以外では効果的でない．整数と浮動小数点の並行処理は浮動小数点演算が支配的なアプリケーション以外では効果がないし，専用のプリフィクスが必要なため，当時のコンパイラは並列実行のための専用コードを生成しなかった．アセンブラの助けが必要である．

88000のビットフィールド命令はビットフィールドの中に対してセット/クリア，挿入/抽出をサポートする．ビットフィールド命令は最近のRISCでの命令拡張では流行になっている．つまり，先祖帰り的な傾向が見られる．

SPARCのタグ演算はデータとポインタに異なるタグを付け，データやポインタに関する不正演算を検出する．これは，LISPやSmalltalkの実装(動的なエラーチェック)に非常に有利である．

▶セマフォ

これらのMPUは，テストアンドセット操作を実現する命令をもち，セマフォをサポートする．i860はロック/アンロック

〔表10〕SPARCの命令セット

算術・論理・シフト命令	
ADD(ADDcc)	加算(と条件コードの変更)
ADDX(ADDXcc)	キャリー付き加算(と条件コードの変更)
SUB(SUBcc)	減算(と条件コードの変更)
SUBX(SUBXcc)	キャリー付き減算(と条件コードの変更)
TADDcc(TADDccTV)	下位の2ビットをタグとみなして加算
TSUBcc(TSUBccTV)	下位の2ビットをタグとみなして減算
MULScc	乗算と条件コードの変更
AND(ANDcc)	AND(と条件コードの変更)
ANDN(ANDNcc)	NAND(と条件コードの変更)
OR(ORcc)	OR(と条件コードの変更)
ORN(ORNcc)	NOR(と条件コードの変更)
XOR(XORcc)	排他的OR(と条件コードの変更)
XNOR(XNORcc)	排他的NOR(と条件コードの変更)
SLL	論理左シフト
SRL	論理右シフト
SRA	算術右シフト
SETHI	rレジスタの上位22ビットをセット
SAVE	呼び出し側レジスタウィンドウの退避
RESTORE	呼び出し側レジスタウィンドウの回復
特殊レジスタ操作命令	
RDY	Yレジスタのリード
RDPSR	PSRレジスタのリード
RDWIM	WIMレジスタのリード
RDTBR	TBRレジスタのリード
WRY	Yレジスタのリード
WRPSR	PSRレジスタのライト
WRWIM	WIMレジスタのライト
WRTBR	TBRレジスタのライト
UNIMP	未定義命令
IFLUSH	命令キャッシュの無効化

ロード/ストア命令	
LDSB(LDSBA)	符号付きバイトロード
LDUB(LDUBA)	符号なしバイトロード
LDSH(LDSHA)	符号付きハーフワードロード
LDUH(LDUHA)	符号なしハーフワードロード
LD(LDA)	ワードロード
LDD(LDDA)	ダブルワードロード
LDFSR	FSRレジスタへのロード
LDCSR	コプロセッサ状態レジスタへのロード
LDF	浮動小数点レジスタへのロード
LDDF	浮動小数点レジスタへのダブルワードロード
LDC	コプロセッサレジスタへのロード
LDDC	コプロセッサレジスタへのダブルワードロード
STSB(STSBA)	バイトストア
STSH(STSHA)	ハーフワードストア
ST(STA)	ワードロード
STD(STDA)	ダブルワードストア
STFSR	FSRレジスタからストア
STCSR	コプロセッサ状態レジスタからストア
STF	浮動小数点レジスタからストア
STDF	浮動小数点レジスタからダブルワードストア
STC	コプロセッサレジスタからストア
STDC	コプロセッサレジスタからダブルワードストア
STDFQ	FQレジスタからストア
STDCQ	コプロセッサキューレジスタからストア
LDSTUB(LDSTUBA)	アトミックなロードとストア
SWAP	レジスタのメモリとのスワップ
分岐命令	
Bicc	整数条件コードによる分岐
Fbfcc	浮動小数点条件コードによる分岐
Cbcc	コプロセッサ条件コードによる分岐
CALL	手続きの呼び出し
JMPL	現在のアドレスをレジスタに退避してジャンプ
RETT	トラップからの復帰
Ticc	整数条件コードによるトラップ
浮動小数点・コプロセッサ命令	
Fpop	浮動小数点命令群
Cpop	コプロセッサ命令群

という命令の組がある．この間にある命令は割り込み受け付け不可となり，アトミック操作を実現できる．88000にはXMEM命令がある．これはコンペアアンドスワップ操作を実現する．SPARCには2種類のセマフォ命令がある．ロードストア無符号バイト命令は，不可分にメモリをリードしてそこにオール1をライトする．SWAP命令はオール1の代わりに特定のレジスタの値をライトする．

これらを比較すると，i860のロック/アンロック機構がセマフォの実現に適しているように見えるが，実際にセマフォを実現するとなると複数の命令が必要であり，三つの間で大差はない．

▶乗除算

これらのMPUの中では，88000のみが乗除算命令をサポートする．i860は浮動小数点命令の乗算があり，これで代用することができるが，除算はない．SPARCには乗除算のためのステップ命令(部分積などを計算する)がある．i860とSPARCは，ますます乗除算が重要になる当時のアプリケーション状況においては不利な立場にあった．初期のRISCの多くに乗除算命令がなかったことは，半ば常識のようになっている．

しかし現在では，乗除算命令をサポートしないアーキテクチャはまずない．たとえばSPARCは，バージョン8で整数乗除算命令が定義された．

このように乗除算命令の有無が比較対象になるということ自体，初期のRISCの特徴をよく表している．

▶分岐

これらのMPUにはいずれも遅延分岐の概念があり，遅延スロットを利用すると分岐のペナルティの60～70％を削減できる．また，遅延スロットを無効化することも可能で，これはコードサイズの減少に役立つ．また，ハードウェアによる分岐予測をサポートする．

▶アドレッシングモード

これらのMPUで共通なオペランドのアドレッシングモードは，「ベース＋オフセット」，「ベース＋インデックス」であり，

常にゼロを値とするゼロレジスタをもっている．これらを組み合わせると，次の五つのアドレッシングモードを実現できる．

- レジスタ：Rx
- レジスタ間接：(Rx)
- インデックス付きレジスタ間接：(Rx，Ry)
- オフセット付きレジスタ間接：offset(Rx)
- 即値

これらはCISCでもっとも頻繁に出現するアドレッシングモードでもある．

88000では，インデックスをデータサイズでスケーリングすることができる．しかし，そのような使用法は人工知能言語や科学技術計算では有用であるが，通常は使われない．i860ではレジスタファイルのリードポート数を節約するためにインデックスアドレッシングはない．しかし，CISCマシンでもインデックスアドレッシングの出現頻度は低いので問題ない．

インデックスアドレッシングは，行列計算を効率的に行える．3次元グラフィクス用途にはあったほうが望ましい．

▶レジスタ

これらのMPUはCISCよりも多くのレジスタを提供するが，実際に何本使用できるかはアーキテクチャによって異なる．この意味では88000のレジスタセットは弱い．整数と浮動小数点に共通な32ビットレジスタが32本あるだけである．i860とSPARCは整数と浮動小数点用にそれぞれ32本の32ビットレジスタを提供する．

実際，レジスタの本数が多いi860とSPARCのほうが88000よりよい性能を達成することがわかっている．SPARCはこれに加えてレジスタウィンドウをサポートする．

6 現在のRISC

RISCにも20年以上の歴史がある．その中で現役として使われているアーキテクチャは，ARM，MIPS，SPARC，PowerPC，PA-RISC，Alphaくらいであろうか．実際には組み込み向けのMPUもRISCアーキテクチャを採用しており，その中で比較的有名なものとしては，SHとV850であろうか．

しかし，それらの命令セットをすべて説明することはあえてしない．どれもバークレーとスタンフォードのRISCを基礎とした発展形にすぎないからである．ここでは，これまでまだ詳しく説明していない，PowerPC，PA-RISC，Alphaのアーキテクチャに関して簡単に説明しておく．

● PowerPC

PowerPCは発表当時，RISCの中でも豊富な命令を備えた「Rich RISC」と呼ばれ，(今ではありふれているが)積和命令やレジスタ値に依存した分岐命令，OS専用命令が注目を浴びた．後々高性能化の妨げになるので遅延分岐は採用しないといったのは有名である．

しかし，PowerPCのアーキテクチャは最初から完成されており，これまでその命令セットには大きな変更はない．PowerやPowerPCの進化は，命令セットをいかに高速化するかというマイクロアーキテクチャの実装方式の進化である．

最初のPowerPCであるPowerPC601は，次のような高速化技術を採用している．

- スーパースカラ
- 命令プリフェッチキュー
- アウトオブオーダ命令発行
- レジスタリネーミング
- ロード/ストアバッファ

これらは大型計算機の技術をいち早く採り入れたものといえる．

● PA-RISC

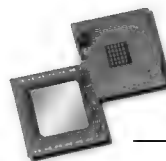
PA-RISC(Precision Architecture RISC)とはHP社のEWSであるHP9000シリーズのアーキテクチャであり，EWSの分野ではかなりの実績をもつ．それでいて，ビット操作命令，ビットフィールド命令，独自機能をサポートするSFU(Special Function Unit)を有し，組み込み制御分野にも適している．

PA-RISCの命令長は32ビット固定長で，140種の命令を提供する．その内訳は，メモリ参照命令，分岐命令，算術論理演算命令，システム制御命令，コプロセッサ命令である．命令の特徴は複合機能を有する分岐命令で，加算と条件分岐，比較と条件分岐，転送と条件分岐の機能を1命令で実現し，1クロックで実行する．そのほかにシフトと加算を1クロックで実行する．さらに，演算と分岐命令には次の命令を無効化する機能がある．これにより，遅延スロットを最適化したコードサイズの圧縮やループプログラムの高速化を行うことができる．

● Alpha

Alphaは当初から64ビットアーキテクチャを提供し，64ビットのロード/ストア命令を基本として命令セットが構築されている．8ビット/16ビットのロード/ストア命令はなく，必要な場合は専用命令でバイトの挿入/抽出を行う(2代目の21164ではこの制限はなくなった)．命令長は32ビット固定で，140種の命令がある．

Alpha AXPの特徴はPAL(Privileged Architecture Library)コードにある．PALコードとは，割り込み例外の処理と復帰，コンテキストスイッチ，メモリ管理，エラー処理など，従来はMPUのハードウェアで処理していた機能を，MPUのハードウェアを直接操作するサブルーチンで実現する．OSやハードウェア構成の違いごとにPALコードを用意することで，基本となるアーキテクチャが異なるシステムにも共通にAlpha AXPアーキテクチャのMPUを搭載できるといわれている．PALコードは，MPUの実装別に定義されるPAL命令と通常命令で構成される．たとえば，1992年に発表された最初のAlpha21064は，次の3種5命令のPAL命令をサポートする．

- 内部レジスタのリード/ライト命令
- MMUを介さないロード/ストア命令
- PALコードからの復帰命令

Column

現在におけるCISC命令セットの意義

● CISCとRISCのプログラムサイズ

RISC命令セットは，MPUの性能を追及してきた成果である．現在ではほとんどのMPUがRISCになっている．それでは，CISC的な命令セットに意義がなくなったのかというと，そうでもない．性能よりもプログラムサイズのほうが重要視されるROMベースの組み込み制御分野では，いまだにCISC的な命令セットが重宝される．このような分野では限られた容量のROMにどれだけ多くの機能(＝命令)を詰め込むことができるかによって価値が決まる．つまり，プログラムサイズ至上主義である．

組み込み制御分野も現在では，Cなどの高級言語を使ってプログラムが記述される．そこで現われる命令機能はかなり定型的である．たとえば，スタックポインタを基準としたメモリ(変数)参照，スタックフレームの生成と破棄，そのスタックフレームへのレジスタの一括した退避と回復，データ型に応じた符号拡張やゼロ拡張などである．これらを複数の基本命令で，パイプライン的に，高速に実行するのがRISCであり，1命令で比較的低速に実行するのがCISCである．明らかにCISCのほうがプログラムサイズは小さい．また，RISCの分岐遅延スロットも，場合によっては命令数を増加させる傾向にあるので，プログラムサイズの観点からはなくてもよい．

● ARMのThumb命令セット

ARMの命令セットは，このようなRISCとCISCの命令セットの中間点をうまくおさえているところに圧倒的な人気の秘密があるのかもしれない．とくにARMの命令長を16ビット化したThumb命令セットは，CISC化の傾向が強い．ARM社は，Thumbコードでは40％の性能低下だが，70％のプログラムサイズに圧縮できるとしている．この一見ネガティブな説明がまかりとおっているということは，約半分の性能になってもプログラムサイズが重要な場面があることの証明であろう．

● MIPS16命令セット

ARMと同様に組み込み制御分野に注目しているMIPSもMIPS16命令セットを定義している．そして，さらにコードサイズを縮小するためにMIPS16e命令セットを定義している．これはMIPS16のスーパーセットで，符号拡張/ゼロ拡張命令，遅延スロットのないジャンプ命令，レジスタの一括退避/回復命令，MIPS32命令の直接実行機能を追加している．ますますCISC色が強くなっている．MIPSライセンシである東芝はMIPS16eを拡張したMIPS16e+を発表し，さらなるコードサイズの削減を目指している．そのおもな拡張機能は，単一ビット操作，ビットフィールド命令，積和命令，飽和命令である．

● ARMのThumb-2

2003年6月，ARM社はThumb命令セットの改良版であるThumb-2を発表した．これは，従来16ビット長のみだったThumb命令セットに32ビット長の命令を混合したものである．ARM本来の32ビット命令とThumbの16ビット命令をモード切り替えする従来方式と異なり，それぞれのビット長の命令の混在を可能とする新しい命令アーキテクチャらしい．しかし，従来の開発ツールを使用可能としているので，命令コード自体は従来の32ビット命令やThumbと互換性があると説明されている．これにより，16ビット命令のみの場合より25％の性能向上になるという．プログラムサイズは32ビット命令の74％になる．

ARM社によれば，性能が25％上がった分，動作周波数を下げられるので，低消費電力化が実現できるとしている．ほとんど詭弁(?!)のような説明ではある．16ビット長と32ビット長の命令を混在させることで，Thumb-2はMIPS16により近くなったといえる．

PALコードによりアプリケーションプログラムの実行とOSの実行が分離されているため，ハードウェアはアプリケーションプログラムの命令セットを高速に実行できるように最適化されている．

SIMD命令/暗号化処理命令

● マルチメディア対応命令

SIMD(Single Instruction Multiple Data)とは，一つの命令で複数のデータを処理することを意味するという，演算方式を表す言葉である．各プロセッサメーカーは命令セットに独自性を出すために，マルチメディア対応や特定分野対応の命令を追加するのに躍起である．

Intelはi386アーキテクチャに**MMX**(MultiMedia eXtention)テクノロジという命令セットを追加した．さらにPentium IIIからは**SSE**(Streaming SIMD Extension)を，Pentium4で**SSE2**という命令群を追加している．AMDも同様に，**3DNow!**という命令群を追加している．

MIPSでは，**MDMX**(MIPS Digital Media eXtension：マッドマックスと発音する)というマルチメディア系の命令セットを追加し，そのサブセットがR5432で実装された．また，単精度浮動小数点を並列実行するための**MIPS-3D**という命令セットも発表され，R20000で実装された．

PlayStation2のEmotionEngineのベクトルユニットに実装されているマルチメディア命令群も忘れてはいけないだろう．これは東芝のTX79コアにも継承されている．

ARMも2000年のMicroprocessor ForumでSIMD命令の追加(v6アーキテクチャ)を表明した．

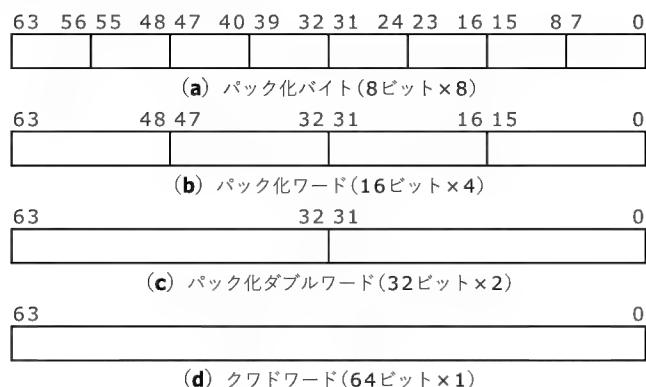
SPARCは，64ビットアーキテクチャのバージョン9でマルチメディア系の**VIS**(Visual Instruction Set)を追加した．それはUltraSPARCで実現されている．

整数だけでなく浮動小数点演算系の強化をしたものには，PowerPCの**AltiVec**もある．

SH-4は最初から浮動小数点のSIMD命令を命令セットとし

〔表11〕MMXの命令セット

オペコード	オプション	実行クロック	記　述
PADD[B/W/D] PSUB[B/W/D]	ラップアラウンド 飽和	1	パック化データの加減算を並列に実行
PCMPEQ[B/W/D] PCMPGT[B/W/D]	一致 より大	1	パック化データの比較を並列に行い，マスクを生成
PMULLW PMULHW	結果が下位 結果が上位	レイテンシ 3 リピート 1	パック化ワードデータの乗算を並列に行い，結果の上位または下位を選択
PMADDWD	16ビットから 32ビットへの変換	レイテンシ 3 リピート 1	パック化ワードデータの乗算を並列に行い，隣接する32ビットの結果を加算
PSRA[W/D] PSLL[W/D/Q] PSRL[W/D/Q]	シフト量がレジスタか即値か	1	パック化データの算術論理シフトを並列に実行
PUNPCKL[BW/WD/DQ] PUNPCKH[BW/WD/DQ]		1	パック化データをインタリーブしながら混合
PACKSS[WB/DW]	常に飽和	1	パック化データを並列に生成
PLOGICALS		1	ビット単位の論理演算
MOV[D/Q]		1	転送
EMMS	実装依存		FPレジスタのタグを空にする

〔図1〕MMXのデータ型

て提供している．SH-5では整数系のSIMD命令も採用されるようだ．2003年6月のEmbedded Processor Forumでは，SH-5のSIMD命令の紹介が行われた．8ウェイのSIMD命令は，たとえばMPEG-4のエンコード時に威力を発揮するという．このように，SIMD命令の採用は花盛りである．

ここでは基本をおさえるという意味で，MMXについて学んでおこう．あらためて見直すと，MMXが提供する機能は，他のプロセッサが採用するマルチメディア命令の機能とほとんど同じなのがわかる．そして最後に，ARMとMIPSのSIMD命令に関して少し言及する．

● MMXテクノロジ

MMXの基本的な考え方は，8ビットまたは16ビットの要素を一つの比較的小さなデータにパックして並列に処理することである(**表11**)．具体的には次のような機能を有する．

▶パックされたデータ形式

MMXでは新しいデータ形式を定義する．マルチメディアアプリケーションで扱うデータの多くは，8ビットまたは16ビットとサイズが小さい．また，マルチメディア処理は多くの隣接するデータ要素を同時に扱うことが多い．MMXでは，これら二つの特色をSIMD処理で実現する．いくつのデータ要素を並行処理すればいいかは，アプリケーションの特性に依存するので，1種類には決定できない．ただ，Intelのプロセッサは64ビットのデータパスをもっているので，MMXのデータ型も64ビットと決められた．具体的には**図1**に示すように，4種のデータ型がある．

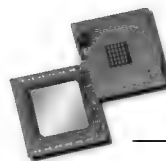
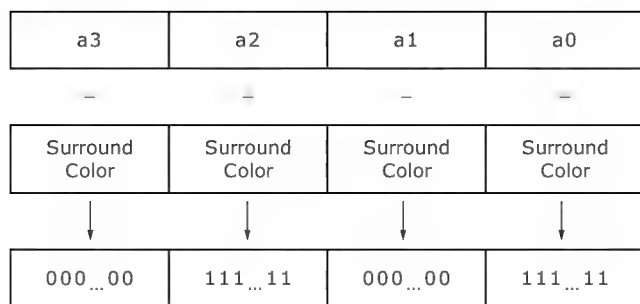
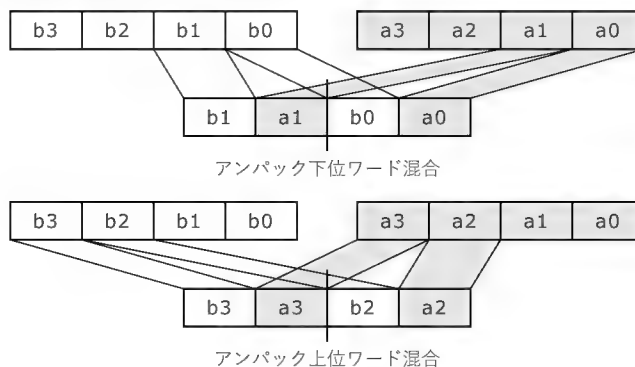
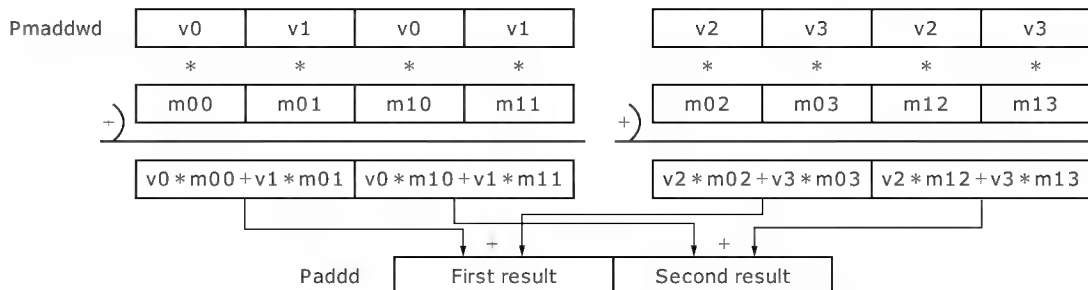
▶条件付き実行

条件によって操作を切り分ける場合，分岐命令を使用することが考えられる．しかし，分岐予測を誤る場合の損失を考慮すると実行速度は遅い．さらに，従来の手法を適用しようとすると，パックされたデータ型をスカラ型(組になってない形式)に変換する必要がある．

これを解決するのが条件転送(条件に応じて転送するデータを切り分ける)である．しかし，このためには三つの独立したオペランド(ソース，デスティネーション，各データ要素に対する条件の組)が必要なので，2オペランドを基本とするインテルアーキテクチャでは都合が悪い．条件転送(条件代入)にはいろいろな実装方式があるが，MMXではマスクつきの代入を採用する．しかし，これには三つのオペランド(ソース，デスティネーション，マスク)が必要である．

そこで，MMXではマスク生成と代入を2段階に分離した．専用の比較命令が各オペランドに対応するビットマスクを生成する．たとえば，比較処理は，八つのバイトがパックされたオペランドに対して，八つの8ビットのマスクを生成する．そのマスクを論理演算と併用することで条件転送を実現できる．**図2**に，四つのワード(32ビット)要素に対する比較操作を示す．

▶飽和演算

マルチメディアで典型的に使用されるオペランドサイズは小さい．たとえば，RGBαという色の各要素は8ビットで表現される．8ビットで256階調の色が表示できる．これは人が認識

〔図2〕ワード型に対するパックトイコール

〔図3〕MMXのアンパック命令

〔図4〕行列-ベクタ演算

できる解像度を超えているが，問題点もある．8ビットでは多くのピクセルの色を蓄積していくと，8ビットで表現できる上限を超えてしまう．デフォルトの設定では，二つの数値の加算結果が上限値を超える場合はラップアラウンドする．つまり，結果が8ビットで表現できなくても，下位8ビットをそのまま値とする．しかし，メディアアプリケーションでは，そのようなオーバフローに対する防御策が必要となる．具体的にはラップアラウンドせず最大値に留まることが望まれることもある．

▶固定小数点演算

メディアアプリケーションでは，フィルタ処理などにおける重み付け係数を扱うために，小数点処理が必要となる．これに対応するために，浮動小数点のSIMD処理を提供することも考えられる．しかし，浮動小数点処理はハードウェアの負担が大きいし，実際のメディアアプリケーションでは10～12ビットの精度で，動的には4～6ビットの範囲が表現できれば十分である．

このような状況を鑑み，MMXでは固定小数点演算をサポートすることとした．固定小数点演算は加減算に関しては整数演算と同一視できるが，乗除算に関しては整数演算を適当にスケーリング（右シフト）しなければならない．これらの機能をサポートしているわけである．

しかし，3D分野でのジオメトリ変換など，単精度浮動小数点の演算精度が必要なアプリケーションもあるのも確かである．これらは，後年，SSEやSSE2で実現される．

▶パックされたデータ型のデータ要素の並び替え

アプリケーションによっては，パックされたデータ内での要素の並び替えや，二つのパックされたデータのマージが必要である．一般的には二つのパックされたデータをオペランドとし，デスティネーションに任意の順序で各バイトを混合することを許すことである．

しかし，これでは実装が複雑になる．MMXでは，アンパック命令により，パックされたデータの要素の並び替えと結合を行うことができる．この命令の動作を**図3**に示す．この命令を使用すれば，ピクセルのパック型の形式をプレーン型の形式に変換できる．

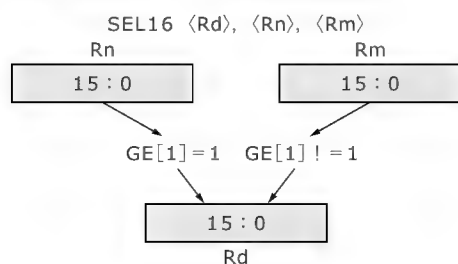
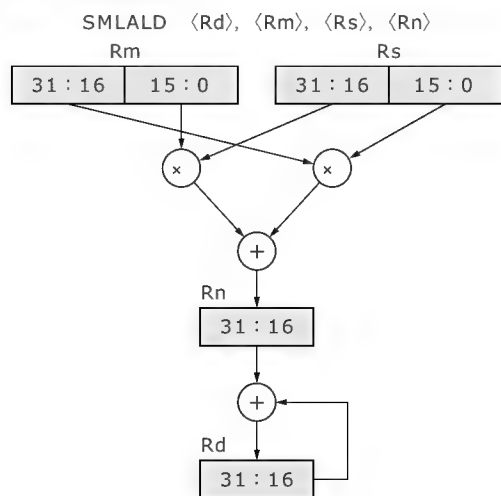
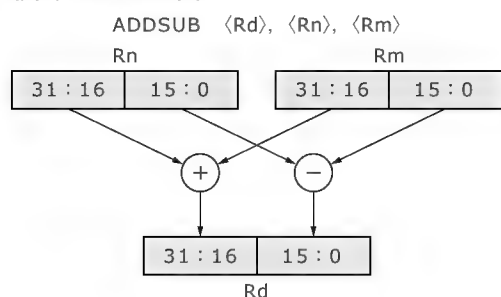
▶積和演算

マルチメディアや通信アプリケーションにおいてもっとも頻繁に出現するのが，積和演算である．これは行列の乗算やフィルタ操作の基本操作として使用される．積和演算を用いる行列-ベクタ操作の例を**図4**に示す．

● ARMのSIMD命令

ARMは2000年秋のMicroprocessor Forumで，既存の命令セットに追加されるSIMD命令の概要を発表した．IntelのMMXと同じく互換性の維持を第一に考え，パイプラインの実行に影響を与えない18の命令を定義した．このため，命令機能は，加算，減算，選択，乗算，飽和に関するものに限られている．その他の命令機能はARMのコプロセッサインターフェースを通じで使用できる．これはStrongARM（XScale）のSIMD命令の実装と同様である．ただし，互換性はない．

ARMのSIMD命令の特徴的なところは，従来のハードウェア資源を利用して命令を拡張したことである．特殊レジスタの

〔図5〕新しい条件フラグを使う選択演算

〔図6〕積和演算の例

〔図7〕SUBADD命令

追加も，ベクタ処理をサポートするALUも追加していない．ただし，SIMD命令のために新しい状態フラグを定義する．それがGE[3：0]で，プログラムステータスレジスタのCPSR[19：16]にマップされている．**図5**はこの条件フラグを使う選択(SEL)演算の動作例である．

SIMD命令では16×16ビットの積和演算をサポートするのが流行であるが，ARMもその例にもれない．**図6**はARMの積和演算を示す．`SMLA{X}D`命令は二つのレジスタの上位16ビットと下位16ビット同士をたすきがけに乗算し，その積を加算する．そしてその結果の上位または下位16ビットをもう一つのレジスタの上位または下位16ビットと加算する(交換処理)．これは，フィルタ処理や複素数の積の計算に有効である．

また，有用なSIMD命令として加算-減算(`ADDSUB`)，減算-加算(`SUBADD`)命令がある．これは，**図7**に示すように，16ビットのデータ同士で行われる．この操作は，FFTやDCTの変換処理に使用できる．

● MIPS-3D ASE

MIPS-3D ASE(Application Specific Extension)は，3次元グラフィックアプリケーションを高速に処理するために，MIPS社が提唱している拡張命令セットである．これは，3次元ジオメトリ処理のために，従来の命令セットに新たな13命令を追加したものになっている．従来の，単精度，倍精度浮動小数点のデータ型のほかに，ペアドシングル(Paired Single)，ペアドワード(Paired Word)というデータ型が新設された．ペアドシングルとは，一つの64ビット浮動小数点レジスタに二つの単精度浮動小数点データを格納するものである．ペアドワードとは，一つの64ビット浮動小数点レジスタに二つの単精度固定小数点データを格納するものである．これらは2ウェイのSIMD方式での処理(要は並列実行)を可能にする．

MIPS社によれば，MIPS-3Dを用いると，もっとも内側の処理ループのコードサイズを30％削減できるので，1秒間に処理できるポリゴン数が45％増加するそうだ．頂点の座標変換での行列の乗算を高速化するために，ペアドシングルリダクション加算命令(`ADDR`)が定義された(**図8**)．リダクションとは行列とベクトル間の乗算の部分的な乗算処理を指すらしい．画像のクリッピングは，ペアドシングル絶対値比較命令(`CABS`)と多重条件コード分岐命令(`BC1ANYnx`)によって簡略化できる(**図9**)．

透視変換には逆数命令(`RECIP1`，`RECIP2`)が使用できる．光源処理には逆数平方根命令(`RSQRT1`，`RSQRT2`)が使用できる．これらの逆数演算はMIPS64アーキテクチャにあるが，より高速に実行できる．

また，ペアドシングルとペアドワード間でデータ変換を高速に行う命令(`CVT.PS.PW`，`CVT.PW.PS`)もある．

追加された13命令の概要を**表12**に示す．

● 暗号処理命令

暗号といえば，従来はICカードやスマートカードの機密保持に使用するものであった．しかし，ネットワークが普及するにつれて，ネットワークを介したデータ転送の暗号処理機能が重要になってきた．従来は外付けのコプロセッサで対応していたが，より高速な処理を達成するため，暗号処理の基本機能をMPUの命令として提供することが考えられている．

本来，暗号は自動化(ハードウェア化)が困難なように構成されるので，命令セットでサポートするのは無謀であるともいえる．しかし，純粋にソフトウェアで記述するよりも20～100倍の性能向上が期待できるので，MPUの特色を出すためには回路規模を犠牲にしても採用する意義がある．

2002年6月，SunはUltra SPARC Vに，暗号処理機能やネットワークのプロトコルスタックの処理機能を搭載すること

〔図8〕ADDR命令の使用例

```
頂点の変換
[X,' Y,' Z,' W ]=[X, Y, Z, W]*(m0  m4  m8  m12)      FP0=[m1  m0]  FP8=[Y | X]
                              (m1  m5  m9  m13)      FP1=[m3  m2]  FP9=[W | Z]
                              (m2  m6  m10 m14)      FP2=[m5  m4]
                              (m3  m7  m11 m15)      FP3=[m7  m6]

                                  ⇩

[X,' Y ]=[X, Y, Z, W]*(m0  m4)    [Z,' W ]=[X, Y, Z, W]*(m8  m12)
                      (m1  m5)                          (m9  m13)
                      (m2  m6)                          (m10 m14)
                      (m3  m7)                          (m11 m15)

行列の乗算を計算する命令列
MUL.PS   FP10,FP0,FP8        → FP10 = [m1*Y | m0*X]
MADD.PS  FP11,FP10,FP1,FP9   → FP11 = FP10 + [m3*W  m2*Z] = [m1*Y+m3*W | m0*X+m2*Z]
MUL.PS   FP12,FP2,FP8        → FP12 = [m5*Y | m4*X]
MADD.PS  FP13,FP12,FP3,FP9   → FP13 = FP12 + [m7*W  m6*Z] = [m5*Y+m7*W | m4*X+m6*Z]
ADDR.PS  FP14,FP11,FP13      → FP14 = [m1*Y+m3*W+m0*X+m2*Z  m5*Y+m7*W+m4*X+m6*Z]
                                    = [m0*X+m1*Y+m2*Z+m3*W | m4*X+m5*Y+m6*Z+m7*W]
```

〔図9〕CABSとBC1ANYnxの使用例

```
頂点が指定領域にあるかテスト
|X| ≦ |W|
|Y| ≦ |W|
|Z| ≦ |W|
```

```
PUU.PS    [W|W]
NEG.PS    [-W|-W]
C.NGE.PS  !(Y ≧ -W)? !(X ≧ -W)?   →条件コード CC0,CC1
C.NGE.S   !(Z ≧ -W)?              →条件コード CC2
C.LE.PS   (Y ≦ +W)? (X ≦ +W)?     →条件コード CC3,CC4
C.LE.S    (Z ≦ +W)?               →条件コード CC5
BC1F   CC0, 範囲外    条件（CC0）不成立なら範囲外
BC1F   CC1, 範囲外    条件（CC1）不成立なら範囲外
BC1F   CC2, 範囲外    条件（CC2）不成立なら範囲外
BC1F   CC3, 範囲外    条件（CC3）不成立なら範囲外
BC1F   CC4, 範囲外    条件（CC4）不成立なら範囲外
BC1F   CC5, 範囲外    条件（CC5）不成立なら範囲外
```

（a）従来方式での計算

```
CABS.LE.PS   (|Y| ≦ |W|)? (|X| ≦ |W|)?
             →条件コードCC0,CC1
CABS.LE.PS   (|W| ≦ |W|)? (|Z| ≦ |W|)?
             →条件コードCC2,CC3 （CC2は真）
BC1ANY4F  CC0,CC1,CC2,CC3,範囲外
          →  条件が一つでも偽なら範囲外
```

（b）新しい命令での計算式

を検討していることを明らかにした．IBMのPower5でも同様の命令の導入が予定されている．

また，暗号処理機能をサポートするのは暗号エンジンの高速化だけでは不十分ということで，暗号を解く鍵や機器に固有のID番号などの情報を格納する特殊メモリ空間や，乱数生成をハードウェア機能として提供するMPUも登場しはじめた．これらの機能は従来MPUの外部ロジックで実現されていたが，セキュリティを強化するためには，その機能をMPU内部に取り込む必要がある．

2003年1月にはTransmetaが従来からのTM5800に，暗号化エンジンを始めとするセキュリティ機能を組み込むことを表明した．具体的には，DESやDES-X，3DESのアクセラレータと，保護されたメモリ領域を内蔵する．

〔表12〕MIPS-3Dで拡張された命令

ニーモニック	処　理
ADDR	浮動小数点リダクション加算．組み同士の加算
MULR	浮動小数点リダクション乗算．組み同士の乗算
RECIP1	逆数．高速近似値．精度的には劣る
RECIP2	逆数．第2ステップ．精度を上げる処理
RSQRT1	平方根の逆数．高速近似値．精度的には劣る
RSQRT2	平方根の逆数．第2ステップ．精度を上げる処理
CVT.PS.PW	ペアドワードからペアドシングルへの型変換
CVT.PW.PS	ペアドシングルからペアドワードへの型変換
CABS	浮動小数点絶対値比較
BC1ANY2F	二つの条件コードのどれかが偽なら分岐
BC1ANY2T	二つの条件コードのどれかが真なら分岐
BC1ANY4F	四つの条件コードのどれかが偽なら分岐
BC1ANY4T	四つの条件コードのどれかが真なら分岐

2003年4月に出荷されたVIA Technologiesの新しいC3(Nehemiah)では**PadLock**と呼ぶセキュリティ機構(ノイズを利用したハードウェアによる乱数発生器と暗号化エンジン)を搭載している．

2003年5月には，ARM社がARM11以降のMPUでは**TrustZone**と呼ぶセキュリティ機能を内蔵することを表明している．これは，Monitorモードという新しい動作モードを定義し，この動作モードでのみ保護したアドレス空間へのアクセスを可能とする機能である．

8 MPUの今後

● RISCの終焉？

現代において，もっとも普及しているMPUはx86(CISC)であり，RISCは終焉を迎えつつあるという見方がある．System InsiderというWebサイトで，元x86アーキテクトという

Massa POP Izumida氏が，「頭脳放談」という連載の第27回「RISCの敗因，CISCの勝因」〔参考文献1)〕で次のように述べている．

①RISC登場の背景は，単純な命令を高速に動かせば高性能が得られるという考えに基づく．これはハードウェアを単純化することから始まった
②しかし，さらなる高速化要求のために大規模な回路（スーパースカラや多重レベルキャッシュなど）が必要になり，回路が複雑になった
③互換性を維持しながら新機能を追加することも複雑化の一因である
④結果，開発に多くのリソース（人的，物的）が必要になった
⑤CISC（x86）は，PCの普及が巨大な市場を形成し，リソースを費やしただけの見返り（利益）が期待できる
⑥RISCはその基盤であるEWS市場の成長が小さく，注ぎ込んだリソースに見合う見返りがない
⑦CISCにアーキテクチャ上の問題があったとしても，注ぎ込むリソース差が性能差（とくにクロック）や価格差に表れ，RISCよりも有利な状況にある
⑧単純に技術要素のトレードオフを論じるならRISCのほうが合理的であるが，現実世界ではきわめて偏りのある「ハンデ戦」で競争が行われている
⑨EWSの存在意義は既存設計資産が使えるということのみで，CAEツールベンダがコスト/パフォーマンスの高いPC＋Linux対応の製品を投入してきている現在では，EWSの将来性は危うい
⑩コンピューティング分野では，性能が上がらないRISCは青息吐息の状態
⑪RISCが全滅かというと否で，組み込み制御の世界には「高速化＝リソース競争」とは違う競争原理が働いているため，RISCの生き残る道はある

かなり的を射た意見であるとは思うが，筆者はMassa POP Izumida氏とは異なり，コンピュータをRISCとCISCに分類して議論するのは無意味と考える．

実際，現在CISCと呼ばれるコンピュータも内部はRISCであり，たいていは命令デコード部と実行部が切り離される構成を採っているので，命令セットの違いはデコードが終了するまでになくなってしまう．結局，高速化を追及すると行き着く先は同じものになると思われる．CISCが勝ってRISCが負けたのではなく，両者が融合したと考えるほうが自然と思う．また，Massa POP Izumida氏は「性能＝動作周波数」と考えているふしがあり，やはり「x86な人」なのだと思う．

● 最近のプロセッサ事情

最近，SoCではおもしろい（?）現象が起きている．プロセッサの性能はバス転送能力のみで決定され，CPUコア単体の性能は全体の性能に寄与しないというのである．極論すればCPUコアは何でもよく，周辺機能として内蔵されるメモリコントローラやPCIコントローラの性能で，全体のシステム性能が決定するというものである．

これは，ある意味真実であろう．しかし，種々のサービスを提供するためには，プロセッサ性能は必要である．実システムでMPU（CPUコア）の性能が支配的であるか否かは，そのシステムで稼動するまではわからない．あるいは，周辺ユニットの動作周波数の都合から，性能はともかくCPUコアの動作周波数が決定されることもある．

つまり，CPUコアの動作周波数はバスの周波数と同じ，または整数倍であることが望ましい．この時点の要旨は，CPUコアの性能ではなく，いかに周辺デバイスとの共同で動作できるかである．性能ではなく，動作周波数が要求される．このような状況ではプロセッサのマイクロアーキテクチャは意味をなさなくなり，動作周波数のみが議論の的となる．これは，性能向上のための動作周波数の向上とは別の次元の話である．

現在では，CPUコア単体の性能がいくら良くても意味をなさない時代になっているのかもしれない．

まとめ

過去から現在に至るMPUの命令セットアーキテクチャを見てきた．アーキテクチャには，CISCとかRISCという区別はあるものの，それらが提供する命令セットにはたいして違いがないことがわかる．MPUができることは今も昔も変わらない．ただ，SIMD命令による並列処理が新たな潮流といえるかもしれない．この傾向は，他社との差別化のために，今後ますます強くなっていくだろう．

また，一つのアーキテクチャを維持するのはたいへんなことである．システムの環境整備には莫大な金額がかかる．いきおい，ハードウェアは従来との互換性を維持しようとし，ソフトウェア（とくにOS）のサポートのないアーキテクチャは滅びていく．Alphaが滅びIA-64が生き延びていく傾向は，それをよく表している．

参考文献

1) Massa POP Izumida，「頭脳放談 第27回 RISCの敗因，CISCの勝因」（http://www.atmarkit.co.jp/fsys/zunouhoudan/027zunou/end_of_risc.html）
2) H. Kaneko et al.,“Realizing the V80 and Its System Support Functions”, *IEEE Micro*, April 1990, pp.56-69
3) R.S.Piepho et al.,“A Comparison of RISC Archtectures”, *IEEE Micro*, August 1989, pp.51-62

なかもり・あきら フリーライタ

多くの箇所の温度を1本につないだセンサで計測する

1線式デバイスによるWebベース多点温度計測

鷲尾英雄

はじめに

手軽に温度を測りたい，温度環境を正確に目に見える形で理解したい．そんな要求に応えるデバイスに出会ったのは，2000年1月発行『トランジスタ技術SPECIAL No.69“1線式バス・システム”』[1]の記事でした．そのころはまだ難しかったLinuxでの温度計測が，現在ではフリーソフトウェアであるdigitempの登場で手軽に実現できるようになりました．

一般の温度計測法で多くの箇所を同時に測ろうとすると，1点あたり2万円程度かかり，さらにこれらをネットワーク経由で計測できるようにすると，もっと高くなります．20点も取れば50万円以上は楽にかかることでしょう．

しかし，温度センサDS18B20（Maxim）の登場により，このような環境が1,000円程度のセンサを1本の線につなぐだけで実現できるようになりました．そこで今回は，データベースを使った多点温度計測システムを，シンプルにして手軽にWebベースで実現します．これにより読者の方が手軽に多点の計測にふれられればと考えています．**図1**が，今回の記事で実現するWebで見る温度計測の結果です．口絵**図A**～**図D**（p.19）が，システムで使う温度センサと，センサをケーブルでつないだ取得機器です．

〔**図1**〕**Webで見る計測結果**

1　目的

この記事では，多点温度計測を行い，Linuxサーバを利用して結果をWebからプロット図として見ることのできるシステムを実現します．また，実際に計測した結果についても述べることにします．使用するプログラムは，digitempとgnuplotで，設定ファイルやシェルスクリプトについては別途作成したソフトウェアを使用します．

なお，これらの環境一式は稿末のWebページからダウンロードできます．また，口絵**図A**～**図D**（p.19）のセンサキットも頒布する予定です．センサはMaxim製で，直接購入もできます．参考文献3）のサイトをご覧ください．

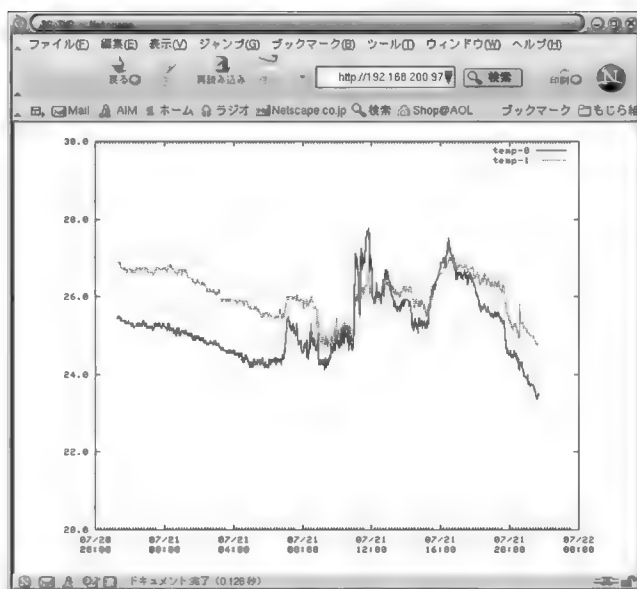
2　概要

- 1線式とは――信号線＋GND（パラサイトモード）

1線式とは，信号線1本で計測することを意味します．通常は信号線とGND，そして電源（＋5V）の3本の線を使います．このうち電源をGNDに結ぶと，信号線から電源を取って動作します．この信号線から計測処理に必要な電気を取るモードを寄生（パラサイト）モードといいます．パラサイトモードでは，A-D変換時に信号ラインから電源を取るので変換の間，信号線を約1秒占有します．これに対して，電源を供給する場合には，このA-D変換の間も信号線が別の目的（他のセンサとの通信など）に使えます．

この電源ラインを使わないパラサイトモードでは，200m先にセンサをつけても問題なく計測します．筆者の社内計測で24点計測をしていますが問題なく動作しています．今回は，手軽なこのモードを使用します．

- 計測原理――個別IDを指定して温度を読む

DS18B20は温度センサですが，通常のセンサと違い，A-Dコ

ンバータと64ビットID(CRC：8ビット＋ID：48ビット＋製品種別：8ビット)，通信機能をもっています．センサというより，小さなプロセッサに温度センサが付いているといったほうが正確です．各センサからデータを読み出すためにはこのIDを指定してA-D変換を要求し，その結果をセンサのメモリから読み出すという順序で行われます．

以上のシステム構成を**図2**に示します．

各センサのデータをすべて取得するには，上記の手順を全センサ分繰り返します．信号線から電源を得るモードでは1センサあたり約1秒の変換時間がかかります．そのため，20点あると約20秒かかることになります．

● 使用するソフトウェア

温度センサDS18B20からのデータ取得からWeb表示まで，次のソフトウェアを使用します．**図3**が処理の流れです．

●digitemp：DS18B20からのデータ読み込み，温度を取得する

digitempの役割を簡単にいうと，次の二つです．

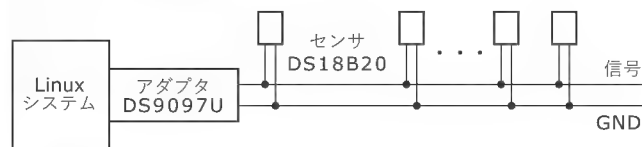
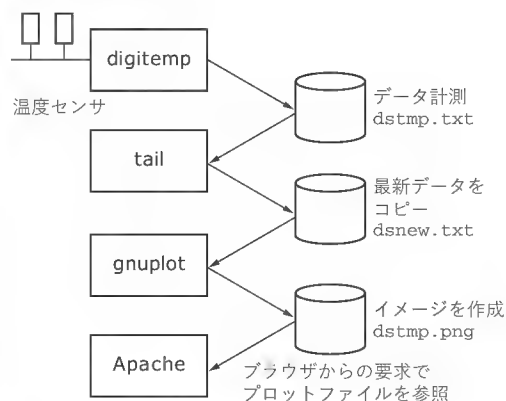
(1) IDの読み込み：各センサの64ビットIDを読み出す

検知順序はつないだ順序ではなく応答順序なので，検出されるセンサの順序はランダムです．

(2) データの読み込み

各センサのIDを指定して温度を読み込みます．

digitempの特徴は，通常のRS-232-C通信プログラムと違って，D9097U(口絵**図D**)を通して1線式のセンサとやり取りする部分にあります．通常のRS-232-C通信ソフトが8ビットのデータのやり取りをするのに対して，ビット単位で制御を行います．たとえば温度を読む場合，スタート条件のビットを送り，その後IDのビット列を送ります．このように同じRS-232-Cポートを使いますが通信方式が異なります．

〔図2〕システム構成

〔図3〕処理の流れ

なお今回は，gnuplotで読みやすいようdigitempのソフトを修正しています．

● tail：ファイル最後部分の取り出し

コマンド形式は，以下のとおりです．

```
tail -行数 ファイル名 > 出力ファイル名
```

このコマンドを使って，保存されている計測データの最新部分を取り出します．1分ごとに計測しているとして1日分は1440分なので，最新1日分を取出すには次のコマンドを使います．

```
tail -1440 dstmp.txt > dsnew.txt
```

● gnuplot：PNG形式でグラフを作成

gnuplotはグラフ作成ソフトです．これだけでも1冊の本が書けるくらいの機能があります．今回は，時間と値のデータファイルからPNG形式のプロット図を作成します．

● Apache：Webサーバ

Webページにプロット図を貼り付けたhtmlファイルを作り，Webブラウザから見えるようにします．Webサーバの設定は標準のままでとくにカスタマイズしません．

3 機器の準備

温度を計測するには，温度センサ(DS18B20)とRS-232-Cアダプタ(DS9097U)そして接続ケーブルが必要です．**図4**が変換アダプタとDS18B20のセンサで，オフィスの天井付近に付けたものです．**表1**にスペックを示します．

4 ソフトウェアの準備

表2のソフトウェアを使用しました．他のバージョンでも，OSに依存する部分がないので問題ないと思います．

● 計測ソフトウェア(digitemp)の準備

(1) ダウンロード

`http://www.digitemp.com/`からダウンロードします．記事執筆時点での最新版である3.2.0を使います．

(2) 展開

適当なディレクトリに`digitemp-3.2.0.tgz`を展開します．

```
tar zxvf digitemp-3.2.0.tgz
```

展開後，`digitemp-3.2.0`のディレクトリが作成されます．

(3) ソースの修正：`digitemp.c`

`src`ディレクトリの中の`digitemp.c`を修正し，gnuplotがそのまま使えるデータを出力する機能と，無限に計測する機能をつけます．

2362行：無限に計測する変更

変更前：`for(x=0; x<num_samples; x++ )`

変更後：`while(1)`

2373行：1行の先頭に年月日時分秒を付ける変更

変更前：

```
case 3:
```

```
    case 2:sprintf(temp, "%ld",elapsed_time);
  変更後：
    case 2:strftime(temp,1024, "%y%m%d%H%M%S",
               localtime(&last_time));
             log_string(temp);
             break;
    case 3:sprintf(temp, "%ld",elapsed_time);
```

(4) コンパイル

digitemp-3.2.0のディレクトリでmakeを実行します．

```
make ds9097u
```

digitempが作成されます．

(5) 動作確認

```
digitemp
```

ヘルプのメッセージが表示されます．接続テストは後述します．

(6) インストール

```
make install
```

/usr/local/binにインストールされます．

(7) デバイスポートにRW権を与える

これは，プログラムを使うのが便利なようにttyポートにRW権を与えています．rootで使うなら必要ありません．

```
chmod +666 /dev/ttyS0  ：ttyS0が誰からも使えます
```

ポート番号は，接続するポートに合わせてください．

5　温度計測とプロット

digitempを使って温度を計測し，その結果をプロットするまでを行います．次のステップのWebで見られるようにすることと自動起動のため，新しくディレクトリを作りそこで以下の作業を行います．ディレクトリ名は，それぞれの環境に合わせてください．この中では以下のディレクトリを使います．

ディレクトリ名：/usr/local/www_temp/，rootで実行

```
mkdir /usr/local/www_temp
chown -R ユーザー名 /usr/local/www_temp
```

ユーザー名は，アクセスする一般ユーザーです．

作ったディレクトリに移動

```
cd /usr/local/www_temp
```

- 温度の計測

(1) コマンドの実行

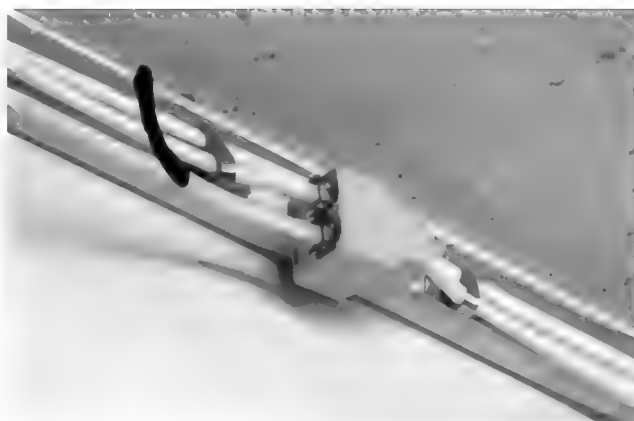
```
digitemp -i -a -o 2 -s /dev/ttyS0
```

図5に実行結果を示します．

二つのセンサが見つかり，.digitemprcに設定情報が書き出され計測結果が表示されます．最後の4行が，年月日時分秒(YYMMDDhhmmss)と各センサの温度です．

(2) 設定ファイルの内容

digitempで-iを指定しているので，設定ファイル(.digitemp)が作成されます．**リスト1**がその内容です．

〔図4〕天井に取り付けた分岐コネクタとセンサ

〔表1〕機器のスペック

温度センサ　DS18B20	
ID	64ビット(センサごとにユニーク)
計測精度	12ビット
計測温度	－10℃～85℃（精度0.5℃） －55℃～125℃(精度2℃) 100℃以上の計測には電源供給が必要
寸法	約5mm四方(センサ部)
電源	パラサイトモードでは電源不要 (供給方法を取る場合3～5V)
RS-232-Cアダプタ　DS9097U	
1線の信号をRS-232-CのTX，RXに直す変換アダプタ	
接続ケーブル	
電話線に使用する6極4芯． ケーブルはシールドツイストペアがベスト． 通常の電話線ケーブル6極4芯が使用でき特別なケーブルを使わないので，手軽で便利．実験ではシールド線を使用して，200mでのパラサイトモードで計測できている	

〔表2〕使用したソフトウェア

OS	Linux2.4.18-1　TurboLinux 8使用
tail	UNIX標準コマンド使用
gnuplot	3.7
Apache	1.3.24

〔図5〕digitempの実行結果

```
DigiTemp v3.2.0 Copyright 1996-2003 by Brian C.Lane
GNU Public License v2.0-http://www.brianlane.com
Turning off all DS2409 Couplers
...
Searching the 1-Wire LAN     * センサを探す
285AD81D0000003A : DS18B20 Temperature Sensor
28FFD81D0000008D : DS18B20 Temperature Sensor
ROM#0 : 285AD81D0000003A     * 64ビットのIDを1つ検知
ROM#1 : 28FFD81D0000008D     * 2目のIDを検知した
Wrote.digitemprc
030721220555  23.56 24.69    * 計測時間と温度
030721220557  23.56 24.69
030721220559  23.62 24.69
030721220602  23.62 24.69
```

〔リスト1〕.digitemprc

```
TTY/dev/ttyS0
READ_TIME 1000
LOG_TYPE 2
LOG_FORMAT "%N%s%C"
CNT_FORMAT "%b%d%H:%M:%S Sensor%s#%n%C"
SENSORS 2
ROM 0 0x28 0x5A 0xD8 0x1D 0x00 0x00 0x00 0x3A
ROM 1 0x28 0xFF 0xD8 0x1D 0x00 0x00 0x00 0x8D
```

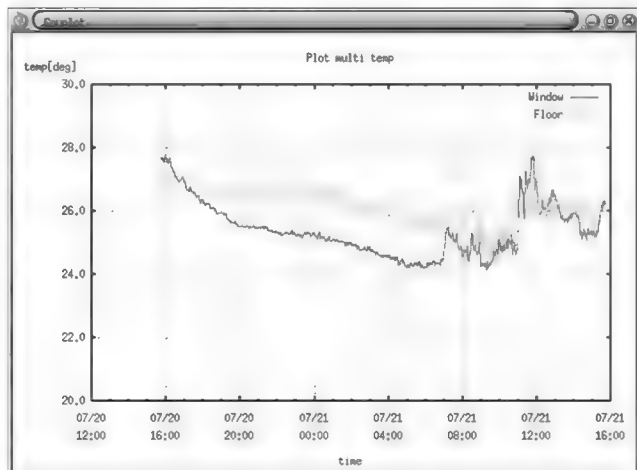
各項目の概要

TTY	：つながっているポート
READTIME	：読み込み時間
LOG_TYPE	：出力タイプ -o 2
LOG_FORMAT	：ログ書式
CNT_FORMAT	：表示フォーマット
SENSORS	：つながっているセンサの数
ROM 0	：センサ0のID
ROM 1	：センサ1のID

〔リスト2〕ds_tst.gp

```
set xdata time
set xtics 14400
set mxtics 4
set title "Plot multi temp"
set xlabel "time"
set ylabel "temp[deg]"
set grid
set format y "%9.1f"
set timefmt "%y%m%d%H%M%S"
set format x "%m/%d¥n%H:%M"
set yrange[20:35]
plot "dstmp.txt" u 1:2 t "Window" w l, pause-1 "hit key"
```

set xdate time	：X軸が時間軸である事を指定
set format y...	：Y軸の表示書式を指定
set timefmt....	：ファイルの時間フォーマットを指定
set format x..	：X軸のラベル書式を指定(時：分)
set yrange....	：Y軸の範囲(20から35まで) 20℃から35℃までを指定
plot ...,	：一つ目のプロット
"..dstmp.txt"	：プロットデータファイルを指定
u 1:2	：1番目の値を時間2番目をY軸とする
t "Window"	：軸のラベルをWindowとする
w l	：線で値を結んだ折れ線とする
,以降	：1番目を時間，3番目をY軸として表示
pause-1 "hit.."	：X上に表示を行いキー入力待ち 入れないとプログラムがすぐ終了する

〔図6〕X画面でのプロット

● 計測結果をファイルに保存

以下のコマンドの実行で，dstmp.txtに日時と値のデータが書き込まれます．

```
digitemp -q- a -o 2 -d 60 -l./dstmp.txt
```

しばらく実行してCtrl+Cでプログラムを停止しdstmp.txtを確認します．-d 60なので60秒に1回データが出力されます．表示するために10行以上はデータが欲しいので10分程度digitempを実行し，dstmp.txtに結果を保存します．

```
cat dstmp.txt
# 年月日時分秒，センサ1温度，センサ2温度(℃)
030721221555  23.69 24.75
030721221655  23.69 24.81
030721221755  23.69 24.75
```

● プロット

gnuplotの設定ファイルを作成します(**リスト2**)．以下に簡単な説明を行います．詳細な説明が必要な場合は，gnuplotの仕様書を参照してください．

以下のコマンドを実行すると，X Window Systemの画面にプロットが表示されます(**図6**)．

```
gnuplot ds_tst.gp
```

7月20日と21日の窓際と室内の温度です．曇りの日だったので温度変化の少ない1日です．

● 日本語表示について

プロットは日本語表示ではありませんが，日本語表示できるgnuplotがWeb上にあります．筆者は試験したことがありませんが，漢字を表示したい方はこちらを使うとよいと思います．

● プロットイメージをPNGファイルに出力するスクリプト

gnuplot dsnew.gpと実行すると，dstmp.pngのファイルが作成されます(**リスト3**)．display dstmp.pngで，X上にプロットを表示します(**図7**)．

6 統合

Webブラウザから見られるように設定します．

● Webページ表示までの考え方

(1) digitempを常駐で動作させdstmp.txtに計測結果を連続的に書き出します．dstmp.txtには，計測開始からすべてのデータが記録されます．

(2) 定期的(1時間程度ごとに)tailとgnuplotを実行し最新の結果をプロット図をdstmp.pngに作成します．tailで行数を指定し，最新の取得期間を指定します．

```
tail -1440 dstmp.txt > dstail.txt
```

1440行＝1行1分として24時間(24 × 60)

(3) 更新されたdstmp.pngをWebページから参照します．

● ディレクトリ構成

(1) PNGファイルディレクトリ

[/tmp] ：PNGプロットデータ

dstmp.png ： Apacheから使えるように/tmpに作成

(2) 結果の保存とプログラムディレクトリ

[/usr/local/www_temp]

dstmp.txt　：計測データ保存ファイル
dsnew.txt　：最新部分のデータファイル
.digitemprc ： digitemp設定ファイル
ds_new.gp　：最新データのPNGを作る設定ファイル
sh_digitemp ： digitempの実行
sh_mk_ds_png ： PNGの更新

(3) Webページディレクトリ

以下を/var/www/html/ に入れます．

Apacheからアクセスされるので，他のユーザーへのアクセス権も与えてください．**リスト4**にds_tmp.htmlを示します．

chmod 755/var/www/html/ds_tmp.html

<img src=...で作成されたPNGのプロット図を参照します．

● スクリプトファイル

(1) sh_digitemp

digitempで計測保存するスクリプトです(**リスト5**)．バックグラウンドジョブとして実行します．

(2) sh_mk_ds_png

プロット図を作るスクリプトです(**リスト6**)．

30分ごとにPNGファイルを更新します．更新時間はsleepの部分を変更します．

(3) 環境設定

/etc/rc.d/initd/httpd start ： Webサーバの起動
cd /var/www/html/ ： Webサーバのルートディレクトリ
ln -s /tmp/dstmp.png .

Webページにdstmp.pngをリンクします．最後の“ . ”を忘れないでください．カレントの意味です．

(4) 参照

http://IP_address/ds_tmp.html ： Webページ

IP_addressは，今回のApacheサーバのアドレスを指定してください．

7　自動起動の設定

Webサーバを自動起動するように設定します．

● Web自動起動の設定

/sbin/chkconfig httpd on

設定の確認

/sbin/chkconfig --list httpd

httpd 0:off 1:off 2:off 3:on 4:on 5:on 6:off

● プログラムの自動起動

/etc/rc.d/rc.localのファイルの最後に，**リスト7**の起動スクリプトを追加します．

killall部分は初期起動では不要ですが，これだけを自動起動のスクリプトとして使うときに，前に動いているプログラム

〔リスト3〕ds_new.txt

```
set xdata time
set xtics 14400
set format y "%9.1f"
set timefmt "%y%m%d%H%M%S"
set format x "%m/%d¥n%H:%M"
set yrange[20:30]
set term png
set output "/usr/local/www_temp/dstmp.png"
plot "/usr/local/www_temp/dsnew.txt" u 1:2 t "temp-0" w l,
      "/usr/local/www_temp/dsnew.txt" u 1:3 t "temp-1" w l 2
```

説明
set term png　　 ：PNGに出力
set output "xxx"：出力ファイルを指定

〔リスト4〕ds_tmp.html

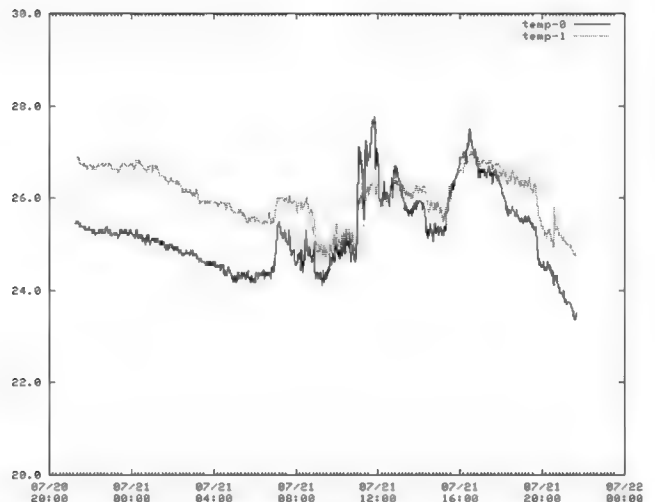
```
<html>
<head>
  <title>DS_TMP</title>
  <meta http-equiv="content-type" content="text/html;
                                        charset=EUC-JP">
</head>
<body>
 <img src="dstmp.png" alt="DSTMP" width="640" height="480">
</body>
</html>
```

〔リスト5〕sh_digitemp

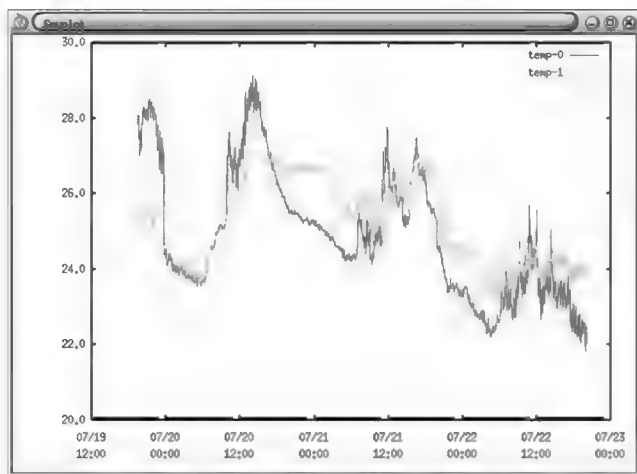
```
#!/bin/sh
/usr/local/bin/digitemp -q -a -o 2 -d 60
                          -c/usr/local/www_temp/.digitemprc
                            -l/usr/local/www_temp/dstmp.txt
```

〔リスト6〕sh_mk_ds_png

```
#!/bin/sh

while[1]; do
tail -1440 /usr/local/www_temp/dstmp.txt
                            > /usr/local/www_temp/dsnew.txt
gnuplot /usr/local/www_temp/ds_new.gp
rm -f /tmp/dstmp.png
cp /usr/local/www_temp/dstmp.png /tmp/
sleep 1800
done
```

〔図7〕PNGのプロット出力

〔リスト7〕rc.localへの追加部分

```
killall sh_digitemp
killall digitemp
killall sh_mk_ds_png
#
/usr/local/www_temp/sh_digitemp&
/usr/local/www_temp/sh_mk_ds_png&
```

〔図8〕3日間のプロット

3日間のプロットで，最初の日より徐々に天気が悪くなっているのがわかる．真中の山が二つあるのは，途中天気が崩れて，また良くなったようす

を止めるのに使います．

- 自動起動の確認

reboot で起動し，ブラウザからプロットを確認します．

`http://IPアドレス/ds_tmp.html` を参照

画面を更新したいときは，更新ボタンを押します．

8　利用と拡張

- 利用

Webで見ながら計測結果を見てみると，1日の温度の変化から天気の変化や室内各部のようすがわかります．

天気の悪い日には，温度の変化が少なく，天気が良いと温度の変化が大きくなります．グラフを見れば，その日の天気を推定できます．また，エアコンの出口などに置いてみると，こんな冷たい風が出ていたのかとか新しい発見があります．部屋に何点も置いてみると，天井と床で温度が違ったり，部屋によって違ったりするようすがいろいろ見えて興味深いものです．冷蔵庫にも入れてみるとドアを開けると温度が上がり，しばらく高い温度が続くこともわかります．

ビニールなどで防水して水に入れると，湯の沸き方もわかります．

- 拡張

簡単なことからはじめると，グラフを1週間や3日などにするには `tail` コマンドの-1440部分を1週間なら1440 × 7 = 10080とすると，1週間のグラフとなります．3日なら1440の3倍です．

1週間や3日，1日，8時間という長さのグラフを作ると大体ようすがわかります．作ってみてください．

3日間のプロットを**図8**に示します．

- 保存

保存データは追記型なので1年以上の長い期間を保存できますが，定期的にバックアップするのがベストです．筆者は何年か前，気象データをテキスト形式で保存していましたが，ディスクのエラーで半年以上のデータを飛ばしてしまい悔しい思いをしています．このようなことがくれぐれもないよう，定期的な保存を心がけてください．

9　製品版の紹介

製品版はPostgreSQLのデータベースを使用しています．そしてプロットルーチンは，軸の変更，ダウンロード，前後へのスクロールなどの基本機能を備えています．統計処理も日報，月報，年報や警報の機能ももっています．また，写真やイラストの上に現在の温度を表示する機能ももっています．

システムとして安定したLinux上にあり，インターネットにつなげばインターネット上から，イントラネットにつなげばイントラネットから，そのデータをタイムリに見ることができるシステムです．ユーザーのシステム開発には，PostgreSQLのデータベースをPHPから使って手軽に作れます．

また，他の計測データが増えても分散データベースによって一つのデータベースとして扱うことができ，将来の拡張にも柔軟に対応します．

口絵**図E**が，USBカメラとDS18B20の21点のセンサを含むシステムです．口絵**図F**がWebで見た画面です．グラフ，現在値，定期的な静止映像，警報履歴，日報，月報，年報，データのダウンロードと必要とされる機能を搭載してインターネットやイントラネットに接続できるシステムです．コンパクトに多点温度計測を実現します．筆者らは，この温度センサを利用して多点記録監視システムを製作しました．口絵**図G**がその機器構成とオフィスの計測結果です．

筆者の会社では，Linuxのベースとして計測システムを作成しており，長期の安定計測システムとして工場や研究所での実績があります．以下はこのシステムを利用した例です．

- オフィスの計測

オフィス各点の温度とグラフが口絵**図G**です．

冬の1日でオフィス全体に張り巡らした各センサからの値は，エアコンを入れてゆっくり暖まっているようすを表しています．

口絵**図H**は1日のプロットで，いちばん上がエアコン出口，以下が机と床の温度です．エアコンは部屋が暖まると温度が下がっています．そして，机の上のカーブは，10時からお昼まで

ゆっくり暖まり，午前中が寒いことがわかります．床は午後3時ごろまでゆっくり上昇し，午後まで床が寒いことを示しています．

● パソコンの計測

口絵**図I**に，パソコン内各部の温度を計測した結果を示します．

パソコンの各部を測ってみると，電源を投入してから各部の温度が上がるようすやCPU負荷の大きいプログラムを実行するとCPU温度が上がることがわかります．また，待機電力も思いのほか大きく，電源を切ることで省エネにつながることもわかります．ここではプロット図を省略していますが，ディスクの温度が外部より約10℃程度高く推移し，ディスクの寿命を短くする50℃以下におさえるために外気を40℃以下にする必要性がわかります．

筆者らのオフィスでは，夏の暑い日もコンピュータを動作させておくと，夏がすぎてよくディスクが不調になります．40℃以上になる暑いオフィスでディスクの温度が上がってしまうからであろうと推測できます．

このように多くの箇所を測るとこれまで見えなかったことがいろいろ見えてきます．そして，それが役立つことを期待しています．

10　Linuxでの多点計測システム

筆者の会社は，宇宙システム開発で培った信頼できるシステムを作る技術やUNIXをベースとして，Linuxを10年以上にわたり利用してきました．今回紹介したDS18B20という温度センサも，コンピュータの集積技術の一つです．わずかなチップの中にセンサA-D変換プロセッサと集積して，しかも安価に手に入れられます．われわれはこれまで多くのデータを手軽に取って有効に利用したいとずっと考えてきました．しかし，これまではとてもたいへんなことでした．それが現在では，使う技術があれば，誰でも使える時代が到来しています．

今回の紹介は温度ですが，温度・湿度や，照度などの情報を手軽に取得できる環境を現在準備しています．われわれが提供する計測環境が，解析とデータの有効利用に結び付けられれば良いと思っています．多くの点の計測からこれまで見られなかった世界が正確に見えてきます．数値に裏付けされたデータは，本当に有効です．このベースに立って，省エネや環境対策，製品の品質，製品の履歴管理など，これから広がる世界にわれわれの技術を生かして行きたいと考えています．

謝辞

digitempの作者であるブライアン氏には，記事として載せるのに問題ないとのコメントを頂きました．この場を借りて感謝いたします．また他のLinuxのソフトウェアについても，作成者に感謝いたします．

参考文献

1）1線式記事：2000年1月発行『トランジスタ技術SPECIAL No.69』，第4章，第5章，CQ出版（株）
2）digitempのWebページ，http://digitemp.com/
3）DS18B20の購入先，http://www.maxim-ic.com/ja/index.cfm
4）gnuplot，http://www.gnuplot.info/

記事関連のソースリストの入手先：http://www.aspect-sys.co.jp/

わしお・ひでお　アスペクト・システム（有）システム開発部
E-mail：washio@aspect-sys.co.jp
TEL：0422-76-7312

組み込みGUI設計の現状とソリューション

第1回 組み込みGUIデザインにおける課題

中山宏之

最近の組み込み機器では高解像度，カラーのグラフィック表示が可能なシステムが増え，これを利用したGUI（Graphical User Interface）を搭載することもハードウェア的には不可能でなくなっている．ところがそのような機器においても，効果的なGUIを搭載した機器は意外に少数派である．このような状況に対する解法の一つとして，ピースクウエアでは「iWinソリューション」というものを提唱し，Windows Embedded OS（Windows XP Embedded/Windows CE.NETの総称）向けに販売開始することとなった．

この連載の第1回では，組み込み機器開発におけるGUI開発の困難さを分析し，どうすれば効果的なGUIを開発できるのかについて解説する．第2回ではiWinソリューションを使用した具体的なGUI開発手順を紹介し，またデザイナの視点からのUI（User Interface）開発ノウハウにふれる．第3回では，iWinソリューションで用いられているATL（Active Template Library）やスクリプティングエンジンWSH（Windows Scripting Host）の利用法など，iWinを実現しているしくみについて，技術的な解説を行う予定である．

iWinソリューションは現時点ではWindows Embedded OSのみをターゲットとしているため，Linuxやその他のOSの個別の事情はあまり登場しないが，連載1回目の内容は，そのようなユーザーにも有用なものと考える．第2回はiWinソリューションの内容に興味のある方，第3回はWindows EmbeddedでのATL / WSHの技術的な内容に興味のある方にぜひ注目していただきたい．　（筆者）

はじめに

● グラフィカルユーザーインターフェース（GUI）の誕生

グラフィック画面を利用してユーザーインターフェースを構築した最初の例は，XeroxのAlto（およびその製品版のStar）ということになっています．それまでにもグラフィック表示可能なディスプレイ装置や，これとX-Yディジタイザを組みあわせて2次元座標を入力可能にしたものなどがありましたが，これらはあくまでも大型計算機の入出力端末としての利用法でした．

じつはこのシステムには，（ハードウェアによる）マルチウィンドウ表示機能とポインティングデバイスとしてのマウスが装備されており，この考え方を取り入れた（Lisaと）Macintoshが，パソコンとしての最初のGUI搭載機種ということになりました．とはいえ，当時の非力なMacintoshのCPU処理能力では512 × 342ドットのモノクログラフィック表示で我慢しなければなりませんでした．

その後，パソコンでのグラフィック表示能力はCPU能力の増大とともに増強され，i486CPUが普及した頃やっとWindows3.xで実用の域に達しました．またMacintoshの世界ではSystem 7でMulti Finderが標準になり，WindowsとともにウィンドウUIで動作する（疑似）マルチタスク環境が実現されました．その後もCPU処理能力は向上し続け，現在パソコンでは3D画面表示もあたりまえにできるようになってきています．

パソコンのグラフィック表示は，当初はキャンバスとしてのグラフィック画面の利用が中心でした．ところがマルチウィンドウをはじめとする高度なGUIプログラムをなるべく簡単に作れるようにするためには，やはり何らかの決まりごとがあったほうがよいということになりました．それで出てきたのがMacintoshのToolBoxでありWindows APIです．これらのしくみを利用してメニューやダイアログ，ウィンドウを簡単に作れるようになってはじめて，効率的なGUIプログラミングが可能になりました．

ただ，その途中の段階ではユーザーインターフェースのいろいろな試行錯誤がありました．たとえば，HyperCardやMacromedia Directorのようなものです．前者はMacintosh用のマルチメディアシステムの先駆けとして（ひょっとしたらWebブラウザの先祖として），後者はCD-ROMベースのいわゆるマルチメディアタイトルのオーサリングツールとして，大いに利用されました．Directorシステムは途中PhilipsのCD-Iシステムを経て現在のDVD Videoのメニューシステムに生き続けているように思われます．

● 組み込み機器のGUI表示は？

パソコンの世界ではGUI（＝ウィンドウ表示）があたりまえになりましたが，組み込み機器の世界ではどうでしょう？　組み込み機器の世界でも，プロセッサの高速化，メモリの低価格化など，ハードウェア的な要因はパソコンと同様です．一方，半導体技術の進歩により，組み込み機器の提供する機能もどんどんと増え続けてきました．複雑な機能を機器に搭載するには何らかのOSが必要になり，複雑な機能を操作するためには表示装置を利用した操作系が利用されるようになりました．

最初は固定パターン表示のLCDだったりしましたが，やが

てキャラクタ表示になり，ものによってはグラフィカル表示になってきました．一方で表示装置は大型化，カラー化，高精細化しており，その上に表示される操作画面がGUIになるのはあたりまえのように思われます．

このようにして，デジカメやDVD録画装置，BSディジタル受信機など，いわゆるディジタル家電の時代には，GUIはあたりまえのものになりつつあります．ところが，本来その優位性が生かせるはずのGUIが実際には使いにくさの元になったり，機器の優位性を表現するために充分活用されていなかったりします．

そのようなことから，今回の連載第1回では，組み込みGUI設計の困難はどこにあるのか，どうすればGUI開発を成功させることができるかについて，説明していくことにしましょう．

組み込みUI開発は難しい？

たとえば，同じWindows APIを利用可能なWindows Embeddedの開発の場合でも，通常のWindowsアプリケーションとは異なる事情がずいぶん存在します．

● 標準的なUIが存在しない

組み込み開発では，（たとえWindows Embedded OSであっても）標準的なWindows UIのような見た目は好まれません．そのようなわけで，組み込み機器のUIは（ある意味Windows開発の効率的な部分を避けて）独自のUIを構築することになります．（たとえばタイマ予約など）特定の場面でしたがうべき標準のようなものもないので，使いやすさや機能の制約を考慮に入れ，すべて一から作り上げる必要があります．これは（少なくとも最初の1回は）開発費がかかることを意味します．プロジェクトによってはカスタマイズが容易であることが望ましい場合もあります（OEMなどを考えている場合）．

● PCとは違ったGUI要件が求められる

PCのプログラムの場合には，ユーザーにある程度の共通の基礎知識を想定し，それをもとにプログラムを作成することが可能です．ところが組み込み機器の場合，対象となるユーザーが普通とは異なる特定ユーザー[注1]である場合もあり，対象ユーザーの特性によってGUIに要求される条件が変わってきます．

UIにおいては，PCよりもわかりやすいものにするため，次のようなことを行います．

- PCよりも選択可能なオプションを少なく
- タスクオリエンテッド（仕事中心）なメニュー
- 裏で実際にやっている複雑さを隠す

タスクオリエンテッドなメニューとは，たとえばWindows XPのエクスプローラに追加されたサイドバーのようなものです（**図1**）．

● UIにおけるさまざまな選択肢

専用機器であるため，いろいろな個別の選択が可能です．場合によっては使いやすさよりも予算の制約を優先しなければいけないかもしれません．

- 画面の大きさ，解像度，カラーあるいは白黒
- 入力デバイス（タッチパネル，外部キーボード，ポインティングデバイス，専用ボタンなど）

〔図1〕
Windows XPのサイドバーの例

選択の自由度の高さが，PCとは違う専用機ならではの特殊な（しかし効率的な）UIを選択するきっかけになる場合があります．逆に，PCよりも小さな画面であらゆる項目を表現しようとして失敗する例もあります．

● 低性能がUIを破滅させる

現在のPCではCPU性能が低すぎるということはなかなかありませんが，組み込み機器の場合，コストの関係上限界ぎりぎりまでCPU性能を落とすことがあります．もし事前の見積もりに反してUIが予想より重かったり，性能の低すぎるCPUを選択してしまった場合に，悲惨な操作性になってしまう場合があります．たとえばJava VM（や.NET Framework？）の上でUIを構築しようとした場合，プロセッサが十分な速度でなかったり，メモリを限界まで減らしてしまったためにウィンドウ描画が遅くなるようなことが考えられます．

● タッチパネルUIの落とし穴

筆者の会社で実際に経験した例ですが，単純にタッチパネルを指で触れてWebブラウザを操作するシステムを開発したとき，次のようなことが起こりました．

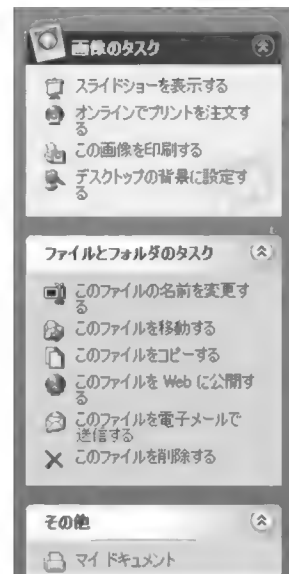
- 通常のWebサイトのデザインは，タッチパネルを指で操作しづらい場合がある

Webサイトはもともと指でタッチパネル操作することを考えていないため，リンク部分が数文字分しかなかったり，ボタンが小さすぎたりすることがあります．このようなときは，マウスやスタイラスペンを用意します．

- 右利きの人は画面左側のメニューにタッチしづらい

マウス使用の場合は問題ないのですが，指を使用してタッチする場合には，右利きの人は画面左側のメニューが心理的に押しづらいようです．

- 右クリックやドラッグの問題

もちろん，タッチパネル画面で右クリックやドラッグをするのが難しい場合があります．パソコンに触れたことがなく，このような操作をしたことがないユーザーがいるかもしれません．これ以外のやりかたで操作できる方法を用意すべきです．

注1：（その機器に詳しくない）一般消費者，ある分野の専門家，愛好家，子供向け，お年寄り向け，肢体不自由者など．

● もしまずいUIを選択してしまうと……

UIの選択を誤ると，次のいずれかの事態に追い込まれるかもしれません．

- 選択したUIを実際に制作したり，あるいは次に変更したりするのが難しくなる
- 本来不必要な開発コストがかかる
- 出来た製品の処理速度が遅く，非効率的なように見える
- ユーザーにとって魅力に欠ける，売るのが難しくなる

組み込みUIの開発手法

前節では組み込みUI開発が本来抱えている設計上の難しさを説明しました．この節でUI開発体制における難しさを説明し，それを解決するための手法を紹介します．

● 開発現場にて——ありがちなやり方

例：さまざまな機能を画面上に表示されたボタンで呼び出せるようにします．

どのようにボタンの機能割り当てを決めましたか？

- 何となく，適当に
- プログラミング上の都合
- 予定されたすべての機能を呼び出せるように，(機能の重要性とは関係なく)なんとかボタンを並べて割り当てた
- デザイナに発注した
  →そちらで適当に割り当ててください
  →格好よく見えるように(使いやすさは二の次)
- とにかくダイアログエディタ(あるいはフォームベース開発ツール)を使って画面だけ作った

▶問題点

- 使いやすさの検討が(まったく)なされていない
- (まれに)UI仕様だけあって機能が本当に実現可能かわからない[注2]
- 使いやすいと思って決めたUI仕様が実際には使いにくい
- デザイナが作った画面イメージに機能を組み込むのに(実際には)無理がある
- スケジュールの問題(少ない開発者でGUI開発まで賄うのは無理！)
- コストの問題(GUI開発にまでお金をかけられない)

▶解決法

● 機器に盛り込む機能の検討

コスト制約やターゲットユーザーの特性を考慮して，どこまでの機能が本当に必要か，またどこまで詳細にユーザーに選択させるかのバランスを再検討します．実際にはUIのデザインのみではなく，機器の機能仕様，あるいはコンセプトに直結する話かもしれません．

● ターゲットユーザーの調査，競合機種の調査

使いやすさの評価は，やはりターゲットユーザーの調査から得るのが最良です．プロトタイプを実際に触ってもらって評価してもらいます．一方，競合機種の調査からは，その機種の良い点，悪い点がわかります．この両方の情報を製品に生かすことができます．

● ストーリーボード，プロトタイプ，ユーザビリティテスト

ストーリーボードを作ることによって，デザイナとプログラマ双方に正しい操作イメージを与えることができます．それをプロトタイプによって確認します．ユーザビリティテストによって，プロトタイプの悪い点がわかります．

● シナリオ作成のすすめ

単一の操作では作業が終わらない場合，シナリオを決めることによってユーザーをうまく誘導できる場合があります．たとえば，実際にウィザードのようなものを作ることによって，一連の設定や作業(たとえば印刷など)をスムーズに行うことができたり，備わっている機能に不足がないか確かめたり，あるいは単にマニュアル作成上役に立つということかもしれません．

● アウトソース，ツールの利用

スケジュール制約やコスト制約を解決するのは，一般的にアウトソースやツールの活用です．グラフィックデザインや優先度の低い部分の機能の作り込みは，アウトソース可能な部分です．逆にメインになる機能や「差別化ポイント」に社内リソースを集中するという考え方も重要です．もし利用可能なツールがあれば，それを積極的に利用することでスケジュール短縮，コスト削減になる場合があります．

● その先のレベルへ……

ここまではある意味あたりまえのことですが，次のいくつかのポイントをふまえることにより，ほかの製品とは一味違った製品を生み出すことが可能になります．

● 認知工学的視点

認知科学の成果[注3]を工学的に応用しようというのが，認知工学の考え方です．たとえば「メンタルモデル」，「ヒューマンエラー」，「プロトコル分析」など，いくつかの知見が応用され効果をあげている例があります．一般にパソコンではデスクトップメタファがよく利用されますが，組み込み系の機器でこれをやって失敗した(？)例(**図2**)があります[注4]．詳細は，出典のWebサイトの内容を読んでみてください．

● デザイナ的視点

実際にグラフィックデザインを仕事にしている人と話してみると，イメージの構築についてのノウハウを垣間見ることができます．このあたりの話は，次回もう少し詳しく取り上げます．

● ユーザーに先進性を感じさせる

これも総合的なデザイン活動の一環です．機器の形，GUIの印象などによってその機器に「先進性」を与えることができる可

注2：競合機種が実現しているから，という理由だけでその機能を仕様書に載せている場合がある．

注3：D・A・ノーマン著，野島久雄訳，『誰のためのデザイン？』などを読んでみてほしい．

注4：General Magic社のMagic CapというOS．

能性があります．たとえば，アップル社のiPodなどです．

- とにかく触ってみたいと思わせる（形状，質感，音，動き）

これも高度なデザイン作業の結果です．何か面白い形，面白い音，面白い（画面上の）動きを組み込むことによって，ユーザーの興味をひきつけることができる可能性があります．たとえば，子供向けお絵かきソフトのKidPixなどです．

- UIのカスタマイズ機能

UIがカスタマイズできれば，ユーザーの興味をつないだり，あるいはOEM時の負担を軽くしたりすることが可能です．このようにするのはたいへんかもしれませんが，もしかしたら（WinAmpのように）ここから評判になるかもしれません．

- ブランディング

ブランドの価値についてはあらゆる書籍で解説されていますが，組み込み機器の分野ではいまひとつ重視されていないように思います．たとえば，表示画面にロゴを貼り付けるだけかもしれませんし注5，1社の製品すべてに何か共通のGUIの印象をもたせることかもしれません．

- 統一されたデザイン

機器の外形，GUIの調子，音などにより機器のかもし出すイメージをコントロールできるかもしれません．たとえば，若者向けにシャープなイメージ，女性向けにソフトなイメージなどです．

- 使いやすいUIとは何か？

ここで，使いやすいUIとは何かということをもう一度考えてみましょう．UIとは結局，組み込み機器にあらかじめ組み込まれている機能を呼び出すための方法です．ユーザーが使いにくさを感じる原因としては，

- ユーザーがやりたいことを指示するのに困難がある（指示の方法が難しい，あるいは指示の方法がわかりにくい）
- ユーザーが期待するだけの性能を機器が備えていない．あるいは必要な機能がそもそも存在しない

などがあります．

一方，UIの作り手側にとっての困難として，次のものが挙げられます．

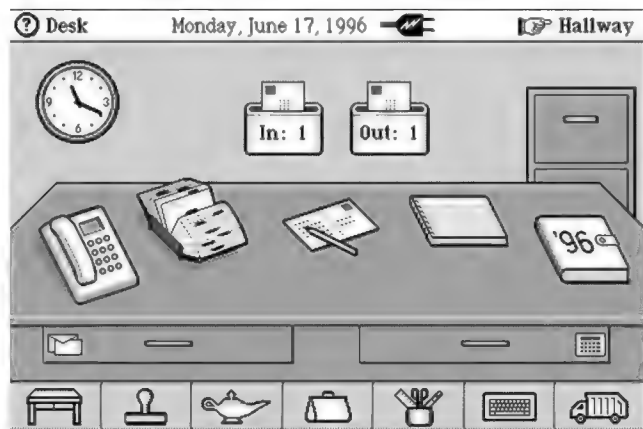
- UI自体の構築に時間と手間がかかる（簡易な方法では，たとえばGUIにした効果が限定されてしまう）
- できたUIが良いものかどうかの評価が難しく，また評価に時間と費用がかかる（決して評価できないわけではないが……）
- どのようなUIが良いかという基準が，対象となるユーザーや機器の分野によってさまざまに変わる

このような困難に対して適切な対応をとれなかった場合，できあがったUIが「破滅的なUI」となってしまうかもしれません．逆にUI（およびそれを含むデザイン全般）を利用して，商品の差別化を実現してしまうことも可能です．

注5：Mac OS 9までのアップルメニューやウィンドウズロゴなど．

注6：Windows CE.NETでHeadlessシステムを構築した場合も，これに似た（GDIのみ使用可能な）状況になる．

〔図2〕組み込み機器にデスクトップメタファを利用すると……

出典：http://hci.stanford.edu/cs147/notes/device.html

- UIデザインで売上が変わるか？

組み込み機器で，ユーザーインターフェースが事実上軽視されている原因は，いくつか考えられます．

- あくまでも機能や性能が第一で，使いやすさの優先順位が低い
- 中間製品（コンポーネント）であって最終製品ではない
- UIを仕上げるまでの期間や予算が足りない
- 顧客の要求事項に含まれていない

もし競合する製品がなければ，そもそもUIのような差別化ポイントも関係ないことになります．開発担当者がいくら使いやすい素晴らしいUIをめざして実現したとしても，それが製品の評価に結びつかないこともあるでしょう．逆に競合製品が存在する場合，エンドユーザーに渡る最終製品（あるいはOEM用製品の企画）の場合，UIをすばやく安価に構築する方法がある場合は，ぜひともUIでの差別化を考えるべきです．組み込み機器ではありませんが，次のような分野ではUIデザインで売上が変わると考えています．

- インターネットのショッピングサイト
- パソコンのアプリケーションソフト
- パソコンのOS（オペレーティングシステム）

Windows Embeddedにおけるさまざまな UI 開発手法

ここまで，UI構築に起因するさまざまな問題点や開発時に考慮すべき事項を考えてきました．それではいざGUIが必要となったときには，どのような開発手法をとることが可能でしょうか？

組み込み分野ではまず，OS標準のウィンドウシステムが存在するかどうかという問題があります．グラフィック表示の可能なハードウェアにOS標準のウィンドウシステムが存在しないという状態は，昔パソコン上のDOSでグラフィックライブラリを使用して画面表示をやっていた頃と同じような状況です注6．この場合，たいていはサードパーティの販売しているグラフィッ

〔図3〕iWinのサンプル画面

クライブラリを使用したり，コンパイラ処理系依存の方法 注7をとる(あるいはフルGUIはあきらめて事実上のCUIで我慢する?)ということになります.

一方，OS標準のウィンドウシステムが存在する場合にも，いろいろな選択肢が考えられます．とくにデスクトップパソコンと同じGUI機能を強みとしているWindows Embedded OSの場合には，開発ツールも開発手法もデスクトップ用のものがそのまま(あるいは似たような方法で)使用可能です．そこで考えられる開発方法には，次のものがあげられます.

- ダイアログベースUI：ダイアログエディタを使ってボタンや入力フィールドなどの「コントロール」と呼ばれるグラフィック部品を配置します．システムにあらかじめ用意してあるWindowsコントロール以外にカスタムコントロールも使用可能です．MFCダイアログベースの場合にはActiveXコントロール 注8も簡単に使用できます.
- フレームベースUI：Visual Basic(VB)から始まったUI構築法です．ツールパレットからボタンや入力フィールドなど(コントロール)をドラッグアンドドロップで配置します．ActiveXコントロールも配置可能です．.NET(Compact) Frameworkのある環境ではVB.NETやC#などを使ってVBと同じようにフレームベースUIが構築可能です.
- スキンベースUI：一世を風靡したWinAmpやその後のWindows Media Playerで採用されているUIです．スキンと呼ばれるグラフィック上に操作可能な部分を定義します．Pocket PC 2002/2003のエミュレータもスキンベースUIとなっています．スキンの下で動くエンジン部分を構築するのはそれなりの手間ですが，エンジン部分がいったんできてしまえば，スキンを変更するのは比較的簡単です.
- ブラウザベースUI：HTMLファイルとグラフィック部品を使用してUIを構築します．表示部をもたずネットワーク経由で設定するようなルータのUIとして多数採用されています(サーバベース)．また，機器上で動作するブラウザ(ローカルブラウザ)を利用して画面表示を全部HTMLで行うようなことも可能です.
- フルスクラッチUI：もちろん，C++やMFCを使用して，すべてのUIを一から構築することも可能です．しかし，(組み込み用としては)あまりにも手間がかかりすぎるため，これまで出てきた方法も含めて普通は何らかの省力化を行うことと思います.
- 特定の処理系に依存したもの：アイデアとしては，Macromedia Flashを全面的に採用したUIや，JavaですべてのUIを構築するようなことも可能ですが，組み込み機器としてはあまりに特殊です.

ダイアログベース，フレームベースのUIでは，ひととおりのUI作成は非常に簡単に終わりますが，それでできるものは通常のWindowsプログラムの画面とまったく変わりません．これは，① WindowsのUIを知る人には非常につまらないものに見えてしまう，② WindowsのUIを知らない人には単に部品が並んでいるだけのように見えて何をすればいいのかわからない 注9，という問題があります.

スキンベースUIやブラウザベースUIは，いわゆるWindows的でない画面を構築することが可能ですが，これを構築するための手間はダイアログベース，フレームベースのものに比べて，どうしても大きなものになってしまいます.

● iWinソリューションとは何か?

さて，ここでiWinソリューションをご紹介します．iWinは，Windows Embedded OSに備わっているブラウザコントロールを利用して作り上げた，ローカルブラウザベースUIを構築するための開発キットです．**図3**は，iWinを利用して実現したサンプルUI画面の例です.

ブラウザベースUIのいちばんの特徴は，UIの見た目のデザイン性にあります．iWinを利用することで，Webデザインの手法を組み込み機器のUIに取り入れることができ，その結果，機器のUIによる差別化が実現可能です．また，ローカルブラウザベースのソリューションは，その拡張性やカスタマイズが容易なところに組み込み機器との相性の良さがあります.

次回は，iWinソリューションを使用した具体的なGUI開発手順，それからデザイナの目から見たGUI開発上のノウハウについて解説します.

注7：Metrowerks社が提供するCodeWarriorのPowerPlantなど.
注8：Windowsの世界で標準的なソフトウェア部品化の方法．またはその部品(コンポーネント)自体を指す．Macromedia Flash PlayerやAdobe PDF ReaderもActiveXコントロールとして提供されている.
注9：「タスクオリエンテッドなメニュー」の項を参照のこと.

なかやま・ひろゆき　ピースクウエア(株)

「VxWORKS」を使ったRTOS技術の基礎と応用

第1回

リアルタイムOS「VxWORKS」の概要

高山 剛

今回から，リアルタイムOS「VxWORKS」に関する技術解説の連載を，6回ほどの予定で開始します．著者はいままで，OSなしの組み込み開発から，VxWORKSを使ったアプリケーションやVxWORKSの移植などに携わってきました．そして，その経験を生かし，VxWORKSの応用方法や解析に関するドキュメントを提供できればと常々考えていました．この機会に，多少とも組み込みシステム開発技術の発展に役立てればと考えています．当連載の筆者は私(高山)だけでなく，WIND RIVER本社のエンジニアや各分野で専門の方々にも参加いただくことを考えています．内容としては，VxWORKSを例に，リアルタイムOSの基礎，ドライバ開発，UNIX系コマンドの移植，Tiny HTTPDの作成，IPv6への対応などのテーマを予定しています．

さて，WIND RIVERは，Jerry Fiddler氏とDavid Wilner氏によって設立された会社です．現在もFiddler氏が会長として，会社の象徴のような存在になっています．

Fiddler氏にはいろいろなエピソードがあります(68K逆アセンブラをコーディングしたのは彼だとか)が，中でも彼がユニークなのは，最初は大学で音楽や写真を勉強していて，プロのギタリストをめざした時期があった話でしょう．コンピュータソフトウェアに興味をもったのも，コンピュータによる作曲の可能性を感じたからだそうです．彼の言葉に，「若い世代で音楽の教育を受けることはコンピュータアーキテクトのための教育として非常に役立つ，音楽の作曲や譜面を書く作業は，実際，小さなフレーズを譜面に落としていき，それぞれの小さなフレーズ内で問題を修正し，それぞれをつなぎ合わせて，さらに全体の調和が一貫するようにつなげ合わせて完成させる．この作業はある意味，単に言語や文法が異なるだけで1種のプログラミングともいえる」というものがあります．筆者には音楽の作曲方法はピンと来ないのですが，逆にソフトウェアの設計やデバッグはどういうものかはわかるので，音楽の作曲の仕方というのがおぼろげながら想像できます．

組み込みシステムの開発も，単純なトップダウンの開発手法では開発できません．トップダウンの設計のみならず，ボトムアップによる設計も必要です．まさに音楽の作曲のように機能，性能，拡張性，コーディングの美しさも考え各モジュール単位で洗練し，モジュール単位で問題解決を図り，モジュール間の接続性や再利用性を考慮しながら，さらに全体の調和が取られていなければなりません．とくにリアルタイム性がポイントの場合は，全体の調和が重要でしょう．

さて，会社設立当時は組み込みシステムのコンサルティング会社として，初期はVRTXやpSOSといったリアルタイムOS(以降RTOSと呼ぶ)に，開発ツールやUNIXライクなI/Oシステムやファイルシステムを加え開発環境を提供し，軌道に乗ると自社カーネル(当時，大学生だったJohn Fogelin氏が開発，Fogelin氏は後にvice president，現在Fellow)と開発環境を提供する企業として発展しました．

ネットワークへの対応

VxWORKSには，Ethernetの登場に合わせていち早くBSDのネットワーク関連部分のコードが移植されました．Berkleyに近いシリコンバレーで誕生した会社だけに，BSDライクなI/Oシステムやコンフィグレーションなどに，エンジニアが大きく影響を受けたようです．移植手法は，BSDのコードにできるだけ手を入れず，VxWORKSに依存する部分とBSDのコードを分離したというもので，そのおかげでBSDの最新コードが取り入れやすい構造になりました．このあたりの事情は，VxWORKSユーザーの交流の場であるVxWorks User Group Email Exploder(誰でも参加可能．http://www-vxw.lbl.gov/vxworks/を参照して参加いただきたい．活発な意見交換がなされている)で，BSDのコードを最初に移植したエンジニアが解説されています．

しかし，ネットワークに対応したことだけが成功要因ではありません．当時は，商用インターネットはなく，Web技術もありませんでした．ネットワークにつながるのは研究者用のUNIXマシンだけだったわけで，組み込み機器がネットワークに対応する必要性はまだなかったのです．では何のためにネットワーク対応を行ったのかというと，それは開発環境のためです．

当時の組み込み開発環境といえば，ICEがあれば比較的高速にOSとアプリケーションをリンクさせてターゲットにプログラムをダウンロードできましたが，ICEは非常に高価でした．そんな状況の中VxWORKSは，ネットワークに対応したこと

で，ネットワークダウンロードを可能にしました．ターゲットCPUをリセットして2，3秒で自動的にダウンロードが完了し，すぐにRTOSが立ち上がる環境は感動的ですらありました．さらにNFSやFTP，RSHを使って部分的なアプリケーションモジュールを動的にダウンロードできるようにしたことも画期的といえます．筆者がVxWORKSに出会う直前の環境は，バグがあったらprintfを入れてmakeでコンパイル，リンク，Sレコードフォーマットに変換して，シリアルで5分かけてダウンロード，よくてフロッピーディスクを抜き差しして行うもので，ダウンロードに時間と手間がかかるものでしたから，目から鱗が落ちたとはこのことでした．

コンパイラ

初期の頃のVxWorksでは，コンパイラはUNIX付属のネイティブコンパイラ，いわゆるccを使っていました．通常はUNIXの `libc.a`とリンクしてUNIX上で動作しますが，`libc.a`の代わりにVxWORKSのライブラリとリンクすることでVxWORKS上で動作させることができました．当時はSunOSやHP-UXが68K系のCPUを採用しており，ターゲットとクロスホスト側のCPUが同じでなくてはなりませんでした．そのため，MIPSを使う場合はDECやMIPS社のワークステーション，もしくはSGIのIRIXを使う必要がありました．

その後GNUのコンパイラGCCが現れます．Intel960，SuperHなど，組み込み市場向けのプロセッサではi960のMPUを使ったホストなどはないので，GNUが使われるのは必然だったかもしれません．

GCCをコンパイラに採用することによって二次的な効果がありました．性能が高かったこともありますが，GCCは一つのソースコードツリーで複数のCPUアーキテクチャをサポートしています．そのおかげで，いろいろなアーキテクチャのコンパイラのバージョンが統一され，移植性が非常に向上しました．たとえば，それまで68K系か86系かという議論がありましたが，プロジェクトが立ち上がったと同時に，そのアプリケーションにピッタリのアーキテクチャを選ぶ道ができたといえます．

この頃からVxWORKSでは，全アーキテクチャのコンパイ

Column 1

リアルタイムOSはなぜ必要か？

プロセッサの性能が高くまた廉価になり，工業機器，ネットワーク機器のみならず，情報家電にも使われるようになりました．最近ではインターネットやUSB，無線LANなどによる接続性をもったり，フラッシュメモリの登場によりファイルシステムをもつ機器も当たり前になりました．従来ならばOSがなくとも，メインループでAの仕事，Bの仕事，Cの仕事と順番に繰り返し処理しながら，割り込みが起これぱフラグを立てて外部事象に対する処理を行うことでアプリケーションを実現でき，通信にしても，メインループから通信モードのメインループに移行すれば実現できました．しかし，システムが複雑になり，通信も1チャネルが複数チャネルになり，Ethernet，USB，シリアルなどへの対応が求められるデバイスが増えると，このようにOSのない環境では指数関数的にシステム構築，メンテナンスの維持が困難になります．このような構造のシステムでは，市販のミドルウェアの利用もカスタマイズなしには不可能です．

また通信時は，サーバ的アプリケーションではシステム稼動中はいつでも応答する必要があり，このようなOSをもたないアプリケーションでは限界があります．RTOSはマルチタスクにより，複雑さをそれぞれのタスクに責任を分担することで，複雑さの指数的爆発を防ぎ，ソフトウェアの再利用性が高まります．またマルチタスクには複数の実現方法があり，LinuxなどのUNIXでは時分割によって各プロセスに一定時間を割り当てますが，実世界の事象と同期したり，ハードウェアをコントロールする組み込みシステムでは，プリエンプティブ(横取り可能)スケジュールが要求されます．これによって，保証された遅延内にアプリケーションの応答が可能になります．

リアルタイムには二つの意味があります．一つは，自動予約システムやWebのように人がストレスを感じない程度のリアルタイム性——これは1秒以内の応答性があれば充分で，RTOSは必要なく，UNIXのような時分割マルチタスクで，充分なCPUパフォーマンスとサーバとの通信速度があれば実現できると思います．しかし，組み込みシステムではメカや実世界の事象を扱うため，はるかに高いリアルタイム性を必要とします．たとえば，ロボットを正確に動作させるためには，決められた数ms～数十μsでモータ制御をしなければなりません．各種センサからのサンプリングを行う場合，msやμsでサンプリングする必要があります．RTOSはこれらの外部事象への対応を，決められたデッドライン内に処理しながら，ネットワークのようにいつパケットが飛んでくるか予測できない事象でも，適切に対応されなければなりません．

しかしマルチタスク，プリエンプティブスケジューリングだけでは，やはり複雑なアプリケーションには対応できません．次のような機能，特性を必要とします．

● RTOSの必要要件

▶機能

●タスク間通信：タスク間でのデータの受け渡し

●タスク間同期

複数のタスク間で協調動作を実現します．

●割り込みハンドラとタスク間の通信，同期

外部事象からの割り込みとタスクの同期，データの受け渡し

ラが同一バージョンのGCCに統一されました．さらに，ホストがHPでもSunでもWindowsホストでも，同じコンパイラが使えました．当然，コンパイラの生成するコードの品質は，コンパイラにバグがない限り同じです．実際に，カーネルのテストはSunでコンパイルされたコードだけターゲットで動作させてテストしますが，Windowsではテストしません．Sunで生成されたオブジェクトとWindowsで生成したオブジェクトを比較することで，RTOSの品質も保証されるわけです．

GCCの採用によるもう一つの利点は，GDBが使用できたことです．GDBはクロス接続可能な設計になっており，さまざまなオブジェクトフォーマットへ対応しているほか，ホストが省メモリでも動作できるようなコーディングになっていたり，高速に起動できるようオンデマンドでデバッグ情報が読み出される構造になっています．GDBとターゲットで動作するRDB（現在ではWDBがRDBを代行）により，RTOSでは必須のクロス開発環境でソースコードレベルデバッグが可能になり，GDBのコマンドラインのインターフェースだけでなく，GUIを備えて操作性も向上し，シームレスな開発環境が完成したわけです．

ちなみに，コマンドライン vs GUI論争が世の中にはありますが，TORNADOでは基本的に両方をサポートしています．コマンドライン/GUIの選択は，エンジニアの好みで使い分けられるべきと考えるからです．エンジニアをデバッグやコーディングだけにいかに集中させるかが開発環境（IDE）の役目です．コマンドラインではディレクトリの移動を意識したり，ブレークポイントに必要な行番号を調べたりといったエンジニアの思考を中断させる要素がありますが，GUIでは思考を中断することなくシームレスな操作が可能です．もちろん逆に，コマンドラインのほうが作業効率が高い場合が多いのは，ご存知のとおりです．

信頼性

信頼性の面でも，WIND RIVERの顧客からアイデアをもらい受け，自動バリデーション（社内ではregression testと呼んでいる）技術を完成させました．当時はtelnetをベースにしたツールでターゲットにloginし，コマンドの発行とその結果をログとして残し，expectと呼ばれるGNUツールで期待する結果と比較するものでした．

●排他制御

マルチタスク下では，複数のタスクが一つの資源（メモリやI/O）に対するアクセスで競合を起こす可能性があります．セマフォ（ミューテックス）でアクセス権限の獲得，解放（VxWORKSではsemTake，semGive）によって，唯一つのタスクから一つの資源へのアクセスが保証されます．

●多重割り込み

リアルタイム性を実現するには，低優先順位の割り込み処理中でも高優先順位の割り込みを許す多重割り込みをサポートする必要があります．

▶特性

●デターミニスティック（決定論的）

応答性はつねに予測可能でなければなりません．RTOSはあらゆるリアルタイムレスポンスやレイテンシに対してワーストケースでのレスポンスタイムを保証しています．たとえば，

　実行可能なタスクが複数あってもコンテクストタイムは一定
　割り込みレイテンシ
　割り込みサービスルーチンの応答時間
　割り込み発生から割り込みサービスルーチンの応答，特定タスクを同期し，特定タスクにコンテクストスイッチされるまでの応答時間

これらのワーストケースを保証し，かつ最短にすることが求められます．

リアルタイムシステムを構築する場合は，前述した機能を必要としますが，実際のアプリケーションでは，ファイルシステム，I/Oシステム，セルフデバッグ機能，浮動小数点演算，ANSIライブラリ，ダイナミックローダ，マルチプロセッサ対応，USB，TCP/IP，TELNET，HTTP，ネットワークファイルシステムなど，広範囲な機能がないと実アプリケーションは構築できません．とくにデバッグ機能は，開発現場だけで使えるICEやソースコードデバッガ，ビジュアライゼーションツールなどのデバッグ機能だけでなく，フィールドでも何らかのデバッグ機能を組み込む必要があります．

近年，もっとも重要と思われるものに，OSの移植性（ポータビリティ）があげられます．SoC，ネットワークプロセッサ，デュアルCPUコア，MPUコアをもったFPGAなど，単にMPUのアーキテクチャの競争からシリコンのプロセスの技術でも競争が起こっているので，MPUの選択は，製品の競争力を決定する上で非常に戦略的な要素を含みます．OSの選択は，どのようなシリコンでも過去のソフトウェア資産が生かせ，多様なアーキテクチャや派生のプロセッサファミリに対応し，ポータビリティが優れているかどうか，という観点で行う必要があります．VxWORKSは一つのソースコードツリー，アーキテクチャに関わらず同一バージョンのコンパイラ，アーキテクチャ非依存のAPI（キャッシュ，MMU，アライメント，エンディアン，BSP）を厳密に定義しています．とくにキャッシュについては，いかなるキャッシュアーキテクチャ（コピーバック，ライトスルー，バススヌープ，ダイレクトマップ，ユニファイドキャッシュ，ライトバッファ，セカンドキャッシュ）でも，キャッシュのコヒーレンシをOSで保証するしくみが必要です．VxWORKSはキャッシュコヒーレンシのメカニズムをもっているので，一つのドライバソースコードであらゆるCPUアーキテクチャ，キャッシュアーキテクチャに対応できます．

現在はtelnetベースのツールでなくWDBを使っていますが，これによって常時OSやミドルウェア，OSのExtension，ドライバ，TCP/IP，ANSI/POSIXライブラリ，浮動小数点ライブラリなど，あらゆるコンポーネントが自動的にテストされているので，OSの信頼性は飛躍的に向上しました．また，バーチャルラボ（何百ものCPUボードを1箇所に集めて集中管理し，Webから電源ON/OFF，予約，シリアルターミナルサーバを介したシリアルコンソールも共有可能で，世界中からボード資産を共有可能にしている），マルチサイト対応のClearCaseを採用するなど，コンフィグレーション・マネージメントを強化し，サポート面やリビジョン管理の面でも品質向上を図っています．

TORNADOの登場

VxWORKSは，カーネルを自社製のWINDカーネルに置き換え，OSの拡張機能としてMMUによるメモリプロテクションの採用（VxVMI）や，マルチプロセッサ環境〔バックプレーンドライバ，VxMP，CORBA（パートナー製品），後にVxFusion〕，C++のランタイムサポート（エクセプションハンドリング，静的ストラクタ，デストラクタ，マングリング，デマングリングなど），ミドルウェアとしてSNMP，SCSI，USB，TFFS，ワイヤレスLANなどを充実させ，さまざまなツールをOSに追加していきました．

このように機能面，開発環境としては非常に発展してきましたが，次に問題となったのは，開発環境を強化しようとすると当然プログラムのサイズが大きくなるというジレンマです．

そこでターゲットに，デバッグに必要なプリミティブだけを実装した小さなエージェントのみを載せてホストと通信しながらホスト側の無尽蔵のメモリを使って高機能なツールを提供できる環境「TORNADO」が開発されました．通信方法もEthernet，シリアル，ROMエミュレータ，後にはICEもサポートすることで，シリアルもEthernetももたないような組み込み機器でも，ターゲットには最小限のエージェントを置くだけで高度なビジュアライゼーションツール（ソフトウェアロジックアナライザと呼ばれるWIND VIEWや特定データをモニタしグラフィカル表示を行うStethoScope，カバレージツールであるCodeTESTなど）が使用できるようになりました．

TORNADOはコードの主要部分のほとんどがTCLでコーディングされてインタプリタで実行されるため，コードの修正，機能追加，可読性に優れ，オープンスペックであるため数多くのパートナーからツールが提供されています．この考え方は他のRTOSにも影響を与えています．しかし，その実装にはTORNADOの登場から8年が経ち（TORNADO以前にもGDBにTCLを組み込み，エージェントをデバッグモニタROMに置いてGDBからデバッグする「μWorks」を開発していて技術的な蓄積があった），何度も見直しがされています．たとえば，シリアルインターフェースが一つしかないハードウェアでも，WDBの通信，コンソール（仮想），リモートファイルシステム（TSFSと呼ばれる：Target Server FileSystem）が同時に使用できたり，ターゲット上のシンボルテーブルとホストのシンボルテーブルが共存する場合，内容を一致させる同期機能も現在では追加されています．

MARS PATHFINDERへの採用

1997年7月4日，NASAの火星探査機「MARS PATHFINDER」にVxWORKSが採用され，宇宙に飛び立った最初の商用RTOSとなりました．当時NASAは低予算で素早く開発する必要があり，自前のカスタムOSでなく商用OSを選択，エアバッグにより着陸時の衝撃を吸収するというアイデアで，見事独立記念日に合わせた着陸に成功しました．MARS PATHFINDERは火星に到着すると，そのスペースクラフト自体がランダーと呼ばれる固定された基地となり，地球からの操作で自由に動き回れるローバー（ソジャーナ）がランダーから降下して通信しながら周辺を動き回り，地球へ貴重なデータを送信しました．

MARS PATHFINDERは，（放射線対策のためと思うが）特殊なMPUが採用されたため，特注バージョンのVxWORKSが搭載され，開発やサポートに従事したエンジニアがロケット発射の際に招待されたと聞いています．報道でもありましたが，MARS PATHFINDERが地球上に撮像写真を送信しなくなった事件がありました（実際には送信されなかったのではなく数日間送信が遅れた）．じつはアプリケーションのタスク間の同期にプライオリティインバージョン（優先度逆転）と呼ばれる，RTOSのしくみでは本質的に起こり得る問題が発生したのですが，この問題はVxWORKSのプライオリティインヘリタンスの機能を使って回避されました（後述）．

プライオリティインヘリタンス

RTOSはプリエンプティブ・プライオリティ・スケジューリング（横取り可能，優先順位優先スケジュール）によりタスクのスケジューリングが行われます．ここでプライオリティインバージョンを理解するために，優先順位の高いタスクと低いタスクが一つの資源を獲得するために，セマフォ（ミューテックスと呼ぶOSもある，セマフォは一つの資源，たとえばI/Oをアクセスする場合，同時に複数のタスクがアクセスしないよう，他のタスクがアクセスするのを待機させる働きをする）による資源の獲得と解放を行う場合を考えてください．

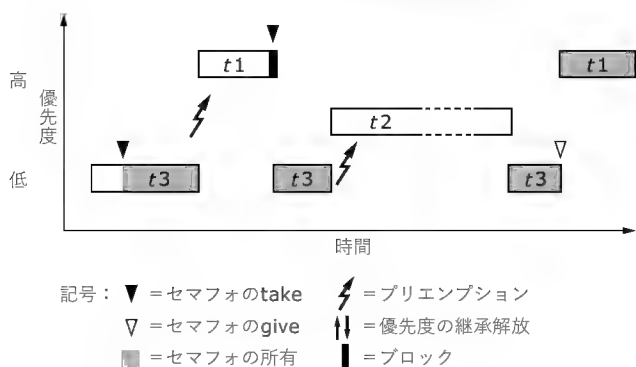
“低いタスクが資源を獲得して実行中，高いタスクが資源を獲得しようとすると，低いタスクが資源を解放するまで，高いプライオリティのタスクは待たされる”

資源の獲得，解放の間のプログラムを必要最小限なコードに

〔図1〕プライオリティインバージョン（優先度逆転）

〔図2〕プライオリティインヘリタンス（優先度継承）

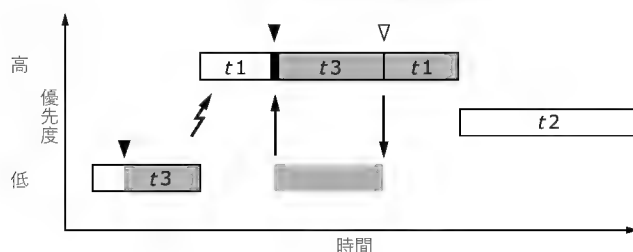
〔図3〕MARS PATHFINDERのタスク

(a)

とどめることでリアルタイム性を確保できます．これはRTOSとして期待どおりの動作です．

ここで，**図1**のように何らかの外部的要因，たとえば割り込みが発生し，中間のプライオリティをもつタスクがReady状態になり動作しはじめると，困ったことになります．低プライオリティタスクは中間プライオリティのタスクが実行を終えるまでずっと待たされることになり，しいては（優先度が）中間プライオリティより高いタスクも待たされることになります．これでは，高プライオリティをもちながら実行が遅延され，リアルタイム性を保証できません．

プライオリティインヘリタンスでは，**図2**のように，低優先順位のタスクを，高優先順位のタスクのプライオリティレベルまでカーネルが一時的に引き上げることで，この問題を回避します．これは，RTOSに必須な手法の一つです．

MARS PATHFINDERの場合は，ローバー（スジャーナ）とランダーが常時通信を行いますが，1553BUSといわれるバスを介してデータを送信するbc_schedと，各種データ（気象情報，土壌分析，高度計などセンサのデータをサンプリングするASI/METタスク，また，それらのデータを集計するbc_distタスクがありました．ASI/METタスクとbc_distタスクは，セマフォを介して通信しています．通常は**図3（a）**のように動きますが，ASI/METとbc_distの中間のプライオリティをもつXタスクが動作したとき，**図3（b）**のように，bc_distがデッドラインに間に合わないため，bc_distとbc_schedのタイミングが合わなくなるのが不具合の原因でした．**図3（b）**のように高優先順位のbc_distより低優先順位のXタスクが優先されてbc_distタスクの実行が遅延され実行されるのが問題の原因です．

このような現象をプライオリティインバージョンといいます．この現象をとらえるのは，実際のカーネルの状態遷移を何らかの方法によってモニタしないと，検出することはほとんど不可能です．NASAのJPLでは，地球上で複製の装置を使って同様の環境を作り出し，OSとアプリケーションに組み込まれたロギング情報を元にプライオリティインバージョンの発生を発見

しました．図3(b)は，プライオリティインバージョンが発生した場合のイメージです．問題の解決は簡単で，VxWORKSのセマフォのプライオリティインヘリタンス(優先度継承)を有効にすることで解決できたようです．図3(c)は，プライオリティインヘリタンスを有効とした場合，どのようにカーネルがふるまうかを示したものです[注1]．

そのバグの修正以降，MARS PATHFINDERは着陸から3か月間で17,000枚の画像を送信し，設計寿命を大幅に越えて，電源の寿命によりその使命を終えました．

RTOSの中には，このプライオリティインヘリタンスをサポートしていないものもありますが，理由として実装が非常に難しいので省略していることが考えられます．8ビットや16ビットではカーネルサイズも大きくなり，アセンブラでスパゲティプログラム化したカーネルでは，実装は困難です．プライオリティインヘリタンスがOSに組み込まれていない場合，MARS PATHFINDERのような高度で複雑なシステムでは，問題を起こし得ます．

とくにI/Oシステムでは，プライオリティインヘリタンスが必要な場合があります．I/Oシステム自体は複雑ではないですが，I/Oシステムの下位層には多様なドライバが呼び出されます．ドライバがハードウェアをアクセスするには，本質的に排他制御が必要でプライオリティインバージョンが起こりえます．しかしI/OシステムをもたないRTOSでは，その重要性が低いと考えられているように思います．VxWORKSはC++のランタイムをサポートしていますが，過去に問題がありC++のランタイムにもプライオリティインヘリタンスの機能の実装が必要なことが判明し，プライオリティインヘリタンスを適用したこともあります．

注1：図3は，説明しやすいように筆者が簡略化したもの．詳しくはYahoo U.S.などの検索エンジンで，bc_distで検索すると，JPLのエンジニアが真相を書いたメールが見つかるはず．

厳しい条件下でのパッチの適用

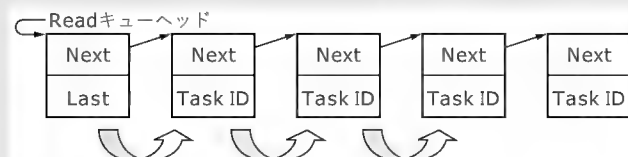
MARS PATHFINDERで使われたかどうかは不明ですが，人工衛星のアプリケーションのデバッグに用いられた方法を紹介

Column 2

なぜ，実行可能(Ready)なタスクが複数あってもコンテクストタイムは一定時間を保証できるのか？

リアルタイムOSの実装は，リスト構造を使ったキューイングの実装が決め手といわれることがあります．たとえばVxWORKSでは，用途に合わせて次のようにさまざまなキューを実装しています．

- ビットマップキュー：Ready状態タスクのキューイング，キューへのノードの挿入，取り出しが固定時間で実現できる
- FIFOキュー(リンクドリスト)：セマフォの待ち状態タスクのキューイング，FIFOに限定されていればもっとも効率的
- プライオリティリストキュー：セマフォなどのプライオリティベースのキューイング，プライオリティをKEYにしてソートが必要とされるキュー
- ヒープキュー：プライオリティリスト・キューを改善し，キューの操作に固定時間を実現

〔図A〕ビットマップキューを用いない場合

- デルタキュー：過去にタイマ用のキューに用いられていた．現在は使用されていないが後方互換性のために存在する

＊　＊

コンテキストスイッチに関係するReady状態タスクのキューイングに用いられているリスト構造は，ビットマップキューと呼ばれる特殊なキューを使っています．これにより，キューイングの際のキューへの挿入時間を固定時間にすることで，Readyタスクの個数に関わらず，固定時間が保証されdeterministic(決定論的)なRTOSとなります．

単純なリンクドリスト構造の場合(図A)だと，Readyタスクの個数がN個の場合，あるタスクがReady状態となると，そのプライオリティに応じてキューに挿入されますが，挿入のノードを探すのにベストケース1回，ワーストケースでN回のリストの探索を必要とし，キューの長さに比例した時間を必要とするためdeterministicが保証できません．

しかし，図Bのようにビットマップキューを用いると，常時STEP1，STEP2，STEP3，STEP4と余分にステップ数がかかりますが，固定時間を実現できます．VxWORKSでは，この辺りはアセンブラでコーディングされているので，図BのSTEP1からSTEP4を20数命令(あるRISCプロセッサの場合)で実現できるので，オーバヘッドはわずかです．

RTOSでは，単に応答が早ければ良いのではなく，deterministicが求められます．リアルタイムシステムでは，固定時間の保証が重要な箇所では，OSが多少余分な仕事をしようと一定の処理時間実現を優先します．

VxWORKSは，レディキューにビットマップキューではなく，ダブルリンクドリストを使うことで，リストの挿入時間を固定時間ではなく，ベストエフォートなカーネルに変更することが可能

します．人工衛星は，もちろん地球上で充分テストされてバグはない状態で打ち上げられたことと思いますが，宇宙という予測外の事態が考えられる環境ではあらゆるバックアッププランが必要でしょう．ハードウェアに故障が発生した場合，ソフトウェアで回避したり，他の人工衛星の代わりに，新しい役割やタスクが必要になるかもしれません．

たとえば，アプリケーションの修正モジュールを，品質が悪く通信帯域の低い衛星や他惑星へ送信することを考えます．この場合，通常のVxWORKSのローダが使えない場合もあります．ローダはオブジェクトファイルをオープンしてTCP経由でSEEKやREADを行います．そのため長距離間でのTCP/IPの確認応答が発生しますが，片道何分何十分という時間のかかる通信にTCPの使用は現実的ではありません（RFCで遠距離対応TCPもあるそうだが）．そこでUDP（もしくは独自プロトコル）を使ってアプリケーションで喪失パケットの処理をし，オブジェクトファイルをいったん人工衛星に転送し，ファイルシステムに格納してからVxWORKSローダによりダウンロードすることになります．

〔リスト1〕サンプルアプリケーション

```
/* testme.c */
testme()
{
   printf ("Clock Rate %d¥n", sysClkRateGet() );
}
```

また，目的の修正モジュールのプログラム（TEXT）やデータ（DATA）だけでなく，シンボル情報，シンボル文字列やリロケーション情報をメモリに展開しなければならないため，設計当初想定より大量のメモリを消費してVxWORKSローダが処理できなくなることもあります．ここで紹介するのは，WIND RIVERのサポートが代替案として提案した方法です．

基本アイデアは，ターゲット側でリロケーションを解決するのではなく，ホスト側で特定のアドレスにアプリケーションをロード可能なように，リロケーションを完全に解決したモジュールを用意する方法です．余分なシンボルの文字列やセクション情報，リロケーション情報を持たない，単なるバイナリファイルを生成しようというものです．

たとえば，**リスト1**のようなアプリケーションのCコードに

です．リアルタイム性の必要性が低く，ネットワークなどのスループットが重要なアプリケーションには有効かもしれません．VxWORKSではコンフィグレーション時に，

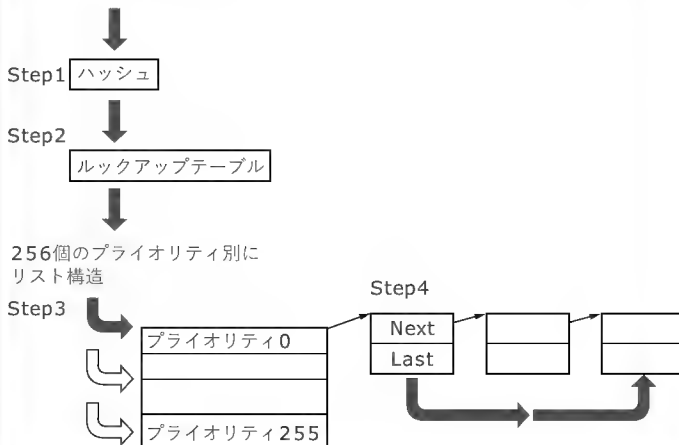
```
    #define INCLUDE_CONSTANT_RDY_Q  /*
           constant insert time ready queue */
```

をコメントアウトすることで変更できます．

ビットマップを使用すると，**図B**のようにプライオリティ0〜255用にキューの先頭と最後尾のリストのアドレスを格納するため，256エントリ×8バイト＝2Kバイトのメモリを使用しますが，Readyキューはカーネルに一つだけ必要なので，許容されるメモリ使用量といえます．

このようにVxWORKSでは，多様なキューをもち，セマフォの待ち行列でFIFOオプションが指定されればFIFOキューを用い，プライオリティベースのキューの場合，プライオリティリストキュー，Readyキューにはビットマップキューというように使い分けています．しかし，キューを実現するユーティリティ関数がばらばらに設計されていた場合，キューのタイプに応じてプログラミングを変えなければなりません．VxWORKSの場合，キューの実装はフレキシブルです．すべてのキューは，同じ数の同じインターフェースをもつようにオブジェクト化されています．つまり，セマフォのFIFOでも，プライオリティベースでも，Readyキューでも，好きなオブジェクトを指定することで特定のキューを使用できます．これを使えば，新しいアルゴリズムで最適のキューを実装して簡単に新しい実装のキューと入れ替えることを可能にしています．

INCLUDE_CONSTANT_RDY_Qの有無でキューが交換できるのも，このしくみがあるからです．もちろん新しいカーネル機能を組み込む際にも，特定のキューに依存したコーディングではなく，このオブジェクトを使用するようにすれば，よりよい実装であることは明白です．キューの実装だけでなく，セマフォ（バイナリ，排他，係数，共有）に関しても，交換可能（トランスペアレント）な「美しい設計」になっています．紹介したのはFogelin氏によるカーネルの設計，実装の一部ですが，彼が20代に行った考え抜かれた実装を通じて，彼が天才と呼ばれる所以を納得していただけるのではないでしょうか．

〔図B〕ビットマップキューを用いた場合

次の行を含むヘッダを追加して，

```
#define printf  (*(FUNCPTR)(0x00020498))
#define sysClkRateGet  (*(FUNCPTR)
                                (0x00020fac))
```

リロケーションが必要な関数コールを絶対アドレスでの関数コールに置き換えるというものです．ヘッダの生成も，UNIXのawkやGNUのnmを使って自動的することが提案されています．

先ほどのtestme.cで説明してみます．まずtestme.cをコンパイルし，リロケータブルなオブジェクトを生成します．BSP上にCのソースがあれば make XXX.oでコンパイルできます．

```
>make testme.o
ccppc -B/host/sun4-solaris2/lib/gcc-lib/
-mstrict-align -ansi -nostdinc -O2
-fvolatile -fno-builtin -fno-for-scope
-Wall -I/h -I.-I/target/config/all
-I/target/h -I/target/src/config
-I/target/src/drv -DCPU=PPC604
-DMV1600
-DTARGET_DIR="¥"mv1604¥""
-c testme.c
```

次にtestme.oのシンボル情報を見てみると，

```
>nmppc testme.o
00000000 t gcc2_compiled.
00000000 T test
U printf
U sysClkRateGet
```

となります．printfがUとなっているため，リロケーション未解決シンボルであるということがわかります．そこで，nmコマンドでVxWORKSのカーネルのprintfやsysClkRateGetが

```
00060730 T printf...
000311d4 T sysClkRateGet
```

であれば，

```
#define printf    (*(FUNCPTR)(0x00060730))
#define sysClkRateGet
                    (*(FUNCPTR)(0x000311d4))
```

を含むヘッダファイルを書いてtestme.cの最初の行でそのヘッダファイルをインクルードすれば，未解決シンボルのないモジュールが生成できます．

アプリーションが大規模ならば，次のようにUNIXコマンドを使って自動的にヘッダファイルを生成できます．

```
>nmppc testme.o | grep U | awk '{print $2}'
                                    > list.$$

>foreach f ( `cat list.$$` )
? nmppc vxWorks | grep " $f" | awk '{print
  "#define " $3 " (*(FUNCPTR)(0x" $1 ")
>> headerFile.h
? end
```

次にheaderFile.hをtestme.c でインクルードして同様に make，nmコマンドで確認すると，

```
>make testme.o


>nmppc testme.o
00000000 t gcc2_compiled.
00000000 T test
```

VxWORKSへの未解決シンボル(U)は存在しないはずです．次にアプリケーションをダウンロードするアドレスを指定し，アプリケーション上のリロケーションを解決してバイナリイメージを生成します．

```
>ldppc -X -N -e test -Ttext 00100000
                          -o testme testme.o
>elfToBin < testme > testme.bin
```

この方法は，関数コールを行う際，直接アドレスで行うためコードサイズが多少大きくなりますが，ターゲット側のメモリが少なく，ターゲット側でリロケーションを行うのに充分な作業メモリがないなど，人工衛星を飛ばしてしまった後で数々の制限のあるなかで採用された解決策でした．

NASAのMARS PATHFINDERへの技術サポートを行ったエンジニアに聞いたところ，MARS PATHFINDERでは火星側のターゲット側のバイナリと地球側の新しいバイナリとの差分を取り，その差分とCRCを送信して，占いプログラムをROMからRAMにコピーし，差分のリスト分を修正してCRCでチェックしてROM(おそらくフラッシュメモリ)に書き戻していたそうです．

ほかにも，nmコマンドやsymAdd関数を駆使して，アドホックな解決策でパッチを適用したケースがあると聞いています．人工衛星やロケットなど，いったん打ち上げられた後，致命的な問題が起こったときには，このようにケースバイケースでさまざまな解決策が取られます．ディジタル家電でも携帯電話でも，いったん大量に市場に出てしまった製品のアップデートにはそれなりのアップデート方法がありますが，どんな機器でも大きな課題だと思います．

執筆時点では無事に2機のマーズ・エクスプロレーション・ローバが打ち上げられ再度火星をめざしていますが，今回もVxWORKSが搭載されています．アプリケーションに問題が出たら，上記のような方法で修正モジュールが送られるかもしれません．

ICEとのインテグレーション

組み込み機器の開発において，Windowsアプリケーションのようにすべてソフトウェアツールで開発できることは理想で

すが，実際はそうはいきません．ハードウェアの設計上の理由によりうまく動かなかったり，初期不良があったり，どうしてもロジアナ，オシロスコープ，ICEなどが必要になります．WIND RIVERは組み込み開発の現場に必要な開発環境をすべて1社でカバーできるようICEメーカーを買収していった結果，現在ではICEと統合開発環境がインテグレーションされました．たとえば，デジカメのようなディジタルコンシューマ製品では，Ethernetやシリアルのような余分なペリフェラルをもちません．そこで現在では，ICEを経由してICEを仮想の通信デバイスとして扱うことで，ターゲットとホストが通信できます．これによりすべてのツールが例外なく使えるようになっています．さらにデバッグ用に最低限のエージェントプログラム(20〜30kバイト)がターゲット内にもてない場合でも，ICEと専用デバッガでOSを直接認識(OS Awareと呼ばれる)してデバッグすることも可能です．

現在のRTOSで求められているものに，ミドルウェアがあげられます．VxWORKSも数多くのミドルウェアやドライバをそろえています．現在では，ディジタル家電向け，ネットワーク機器向けにミドルウェア，ツールをバンドルしてプラットホーム化が進められています．マーケティングもマーケットセグメントに分けられ，市場が要求する最先端のミドルウェアについては，世界で競争力のあるパートナと組んでインテグレーションに力を注いでいます．

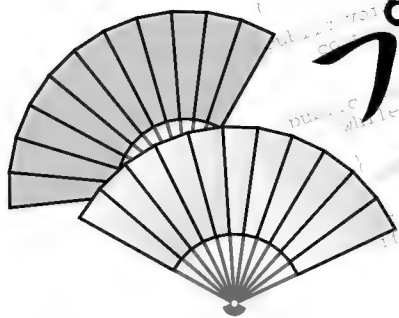
VxWORKSの開発元であるWIND RIVERの歴史を振り返りながら，RTOSの成長の歴史を振り返ってみました．最近，RTOSを使い始めたエンジニア，RTOSと開発環境って何なのだろうと思っていられた方に，多少なりとも歴史から入ることで，理解のお役に立てばと思います．

これからは，Linuxなどのコミュニティベースの OSとの競合は多くなりそうですが，1社でコアの技術をすべてもっている企業は，次世代への対応，技術革新のスピードは非常に早いです．とくに，技術的に行き詰まっても，すぐにスクラップ&ビルドして根本の設計から作り治すこともできます．この場合，既存のユーザーに互換性の面で迷惑がかかりますが，同時にマイグレーションパスも提供して最小の犠牲で最大の技術革新も可能です．このあたりがコミュニティベースのOSに対する有利さといえ，今後もVxWORKSのような商用ベースのRTOSはオープンソースのOSと競争原理でさらに発展し，ユーザーにとっては歓迎すべき状況を形成していくことでしょう．

今後も，VxWORKSをご存知ない方にも読んでいただけるよう，基礎的なこともしっかり解説していきます．近年は，インターネットやワイヤレスLANなど，コネクティビティの必要性からVxWORKSを採用する理由の一つになっています．そこで次回は，VxWORKSでのネットワークをテーマに解説していく予定です．

参考文献

1) Customer Profile JPL/Pathfinder
http://www.WIND RIVER.com/customers/profiles/jpl_pathfinder.html
2) Mike deliman, WIND RIVER, “ how to update SW in space ”
3) The 18th IEEE Real-Time Systems Symposium Keynote address by David Wilner, John Fogelin, “ QUEUE ”

たかやま・たけし ウインドリバー(株)

宮坂電人

第7回

アンチパターンの基礎

正当な法則と裏の法則

プログラミングに限った話でありませんが，この世の中には「正当な法則」と呼ぶには抵抗がある「裏の法則」とでもいうべき法則があります．たいていは冗談であったり，悪意がこもっていてまともに取り上げるのに値しないものだったり，まともに取り上げようものなら，取り上げた人の人格が疑われそうなものもあります．筆者の個人的な意見ですが，ウケねらいや悪意があって取り上げるに値しないと判断する基準は，

① 具体的な個人名や固有名詞(注：アルファベットの頭文字に置き換えていたり，伏せ字を使っていても，読む人が読めば正体がわかるものを含めて)を取り上げ揶揄(やゆ)したり攻撃的な表現があるもの
② 本人の趣味嗜好に偏っていて客観的な評価がなされていないもの
③ 一般化する努力を怠っていて単なる感想文や願望にしかなっていないもの
④ 一部の印象的な現象を拡大して取り上げ，それがさも一般的であるかのようにミスリードさせるもの

あたりだと思います．①のパターンはネット上でよく見かけますが，立場の弱い個人を攻撃することにどんな意味があるのか，理解に苦しみます．なかには自分が勤めている会社(あるいは退職した会社)をコキおろすものもあります．これは法則ではなくて，ただの文句や個人的な復讐/うさばらしとしか思えず，こんなものを時間をかけて読む意味があるのかと途中で放棄することさえあります．

しかしときには，やはり取り上げるべきと判断するものもあり，一概に否定できないのが困ったところです．たとえば「マーフィーの法則」などはそうでしょう．これはかなり悲観的な物の見方を，自虐的な笑いに包んだことで有名な法則です．しかし，さきほどの②と④のパターンにハマっている部分もあり，まともに取り上げるべきか疑問をいだいてしまう法則です．とくに致命的なのは，個人の主観に左右されていて，それによって判断をミスリードさせる可能性の高さです．ものごとを楽天的にとらえる人ではなく悲観的にとらえる人には当てはまるかもしれませんが，たいていは杞憂であったり本人の神経症的(?!)な資質に訴えるものであったりするので，まともにとらえると判断を誤らせることもあります．もちろん物事は悪いことが必ず起きますが，良いことも必ず起きるので，前者ばかりに注目すると，客観的な評価ができなくなります．新聞に不幸な話や大事故の記事が載るからといって，世の中に不幸な状況が蔓延しているとか大事故ばかり起きているわけではありません．幸福になった話や世の中が平穏無事であることを説く記事ばかりだと，おそらくそんな新聞は誰も読まないだろうし，嫉妬心がわき起こって不愉快になるかもしれません．テレビで芸能人の結婚話より離婚話が良い視聴率が取れるのも同様の事情でしょう．

となると法則は，嫉妬深い人間や自分の趣味嗜好に偏って客観的でない人間には作り出せないのか，となるでしょう．ある意味そうですが，この世の中に嫉妬心がまったくない，自分の趣味嗜好を無視して客観的になれる人間というのは，ほとんどいないと思います．しかしそれでも，まともな法則は生み出されますし，それは統計的な手法を利用することで可能だと思います．さきほどの④の「一部の印象的な現象を拡大して取り上げ」る傾向は，およそ人間なら誰でもが陥りやすい傾向ですし，それは人間がもつ動物的な防御本能にもとづくものであるため，根絶することは不可能に近いでしょう．しかし現象を多数調べて統計を取ることで，それが一部なのか普遍的なのかを導き出すことは可能です．

なかには統計そのものを法則にした例があります．有名なものでは「パレートの法則」，「ハインリッヒの法則」があります．

パレートの法則

別名「8対2の法則」とも呼ばれる法則です．1897年にイタリアの経済学者ヴィルフレード・パレートが発見した法則です．彼は当時のイギリスにおける所得と資産の分布を調査しました．すると所得と資産が一部の人たちに集中していることに気づきました．20%の人数の金持ちが，全体の富の80%を有する状況だったのです．後に，このような「全体で20%のXXXが，YYYで全体の80%を有する」という状況がほかにも確認されたことで，パレートの法則は一般的な法則として経済以外でも適用さ

れるようになりました．

プログラミングでも「全体で20%のコードが，処理時間で全体の80%を有する」とか，「全体のコードをチューンナップしなくとも，20%のコードをチューンナップすることで実行効率が上げられる」，というのを聞いた読者もおられるでしょう．ただし，パレートの法則を扱う注意点として，この法則は統計した結果を示しているだけにすぎないということがあります．にもかかわらずパレートの法則は別名「最小努力の法則」で，20%の力を注ぎ込めば80%の成果が得られるかのような極論を説く人さえいます．なかには20と80という「字面（じづら）」にだけ着目し，80＋20＝100という意味だと誤解する人までいます．20はXXXの事象を指し，80はYYYの事象を指していることに気づかなければなりません．取り扱う局面によっては20%のXXXが60%のYYYかもしれないし，10%のXXXが80%のYYYかもしれないからです．

ハインリッヒの法則

アメリカの労災保険会社に勤めていたH.W.ハインリッヒが，50万件以上の労働災害事例の分析をした結果，

- 1件の重大事故（accident）がある場合
- 29件の軽い事故（incident）があり
- 300件の事故とまではいかないがヒヤリとする事態（irregularity）がある

という傾向があることに気づきました．このことから，重大事故を減らすためには，それよりも数の多いincidentやirregularityに気付き，それらを対策すべきというのが，この法則から導かれる教訓です．

ハインリッヒの法則は，さきほどのパレートの法則よりは現代に近いせいか（ハインリッヒの法則が知られるようになったのは1930年代以降），より正確で信憑性が高いと思われます．しかし，これも使い方を間違えると，あやしげな法則に成り下がる危険性を秘めています．たとえば，

- 1人のC++プログラマがいる場合
- 29人のC++プログラマもどき（C++をわかっているつもりだが正確にはわかっていない）がいて
- 300人のC++を学習したが途中であきらめた人たちがいる

という事態があるのでC++の学習は難しいのだ，というジョークがあります（あながちジョークとも思えない怖さがある）．

注意が必要な「アンチパターン」

今回紹介しようとする**アンチパターン**も，取り扱いを誤ると，たちまちあやしげな法則に成り下がる危険性を秘めています．というのもデザインパターンの考えから生まれたものの，デザインパターンよりもわかりやすいし，それどころか，ある種のユーモアをたたえているためトッツキがいいのです．しかし，トッツキがいいもの＝ベストなもの，とは限らないのが困ったところです．早とちりしたり自分に都合のいいように曲解しやすい危険性を秘めています．

アンチパターンとは何であるかを理解するには，それが誕生する元となった“パターン”や“デザインパターン”を知っておくと，よりわかりやすいと思います．デザインパターン，さらにはもっと広い意味でパターンといった場合，「成功する定石」や「成功するパターン」を意味します．にもかかわらず適用を間違えることで，「失敗する定石」や「失敗するパターン」，すなわちアンチパターンになる場合もあります．これはパターンそのものに罪があるのではなく，パターンの使い方を誤っていたり，使うべき局面ではないのに無理に使ったなど，パターンを使った人に罪があるわけです．また，これとは逆のケース，成功するパターンを利用しなかったり，無節操な開発パターンによってプロジェクトが地獄にはまりこんでいる場合もあります．

どちらの場合も共通するのは，どうやら「失敗するパターン」が存在することです．そうした失敗のパターンである，アンチパターンを集めた研究成果が*AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*（William J.Brown，Raphael C. Malveau，Hays W. McCormick III，Thomas J. Mowbray共著，John Wiley & Sons）という書籍にまとめられ（詳細は`http://www.antipatterns.com/`を参照），日本語訳が『アンチパターン―ソフトウェア危篤患者の救出』（翻訳：岩谷宏，ソフトバンク，ISBN4-7973-0758-7）というタイトルで出版されています．

ここからは，日本語訳された同書の内容を紹介しましょう（正確には最初の翻訳版で紹介する．現在出回っているのは新装版ISBN4-7973-2138-5だが内容はほとんど変わっていない）．ちなみに同書は400ページ近くにもなる大著ですが，その一部を紹

Column1

圧力

アンチパターンの本でわかりにくい考えとして“圧力（Forces）”があります．これは，

- 設計の際の選択や決定に影響を与える，一定の状況内での動機付け要因

あるいは，

- 意思決定の方向と内容を決める動機

と説明されています．実際に圧力を与える主体は，開発者個人から会社や業界レベルにいたる広い範囲に渡ります．狭い範囲や特定の問題状況のみで存在する圧力を“垂直圧力（Vertical Forces）”と称し，その反対に，広い範囲や複数の問題を横断して適用される圧力を“水平圧力（Horizontal Forces）”と称します．さらに水平圧力の中でも，ソフトウェアのアーキテクチャや開発に遍在するものを“中心圧力（Primal Forces）”と称します．

介しつつ，筆者なりの解説を加えていきます．

- 「第1章：パターンとアンチパターン入門」

この章では，パターンとアンチパターンについて入門的な紹介をしています．現時点ではソフトウェア開発プロジェクトが六つあるうち五つは失敗とみなされ，約1/3は途中で中止し，残り2/3も予定していた金額や時間を倍以上費やして，ようやく完成したという実態が報告されています．また完成したものは「おんぼろ煙突システム」注1となり，後から補修や拡張がしにくい構造になっています．

何がいけなかったのでしょうか．技術力が稚拙だったり，最新の技術を導入していなかったせいでしょうか．それが原因かもしれませんが，アンチパターンでは必ずしも第一原因と考えないようです．それどころか最新技術を導入することがアンチパターンになり得ることを示唆します．昔から今にいたるまで構造化プログラミング，人工知能，ネットワーキング，オープンシステム，並列処理，オブジェクト指向，フレームワークなどが流行技術となり，これらを導入するとたちまちすべてが解決する「銀の弾丸」として宣伝されましたが，はたして宣伝どおりだったでしょうか．実際は「セールスマンの売り口上」だったのではないでしょうか．

ソフトウェア開発だけに限りませんが，技術問題だけに注目してしまうと根本原因から目をそらしてしまい，「治療」が遅れてしまいます．それはまるで，度のすぎた飲酒で身体に障害が出ているのに飲酒をやめず奇跡的な新薬を探しているようなものです．さて，ここでアンチパターンの定義を簡単にすませましょう．それは，

- 必ず否定的な結果に導く開発方式，しかも一般的によく見られる否定的な開発方式を記述するある種の文献形式

ということです．「必ず否定的な」に注目してください．アンチパターンを採用すると，確実に地獄へ行けるのです．しかし地獄へ行ける方法がわかったからといって，それが何の役に立つでしょうか．われわれが知りたいのは天国へ行ける方法なのに，という反論もあるでしょう．かつてニコロ・マキアヴェッリは，その著作の中でこう述べています．

- 天国に行くのにもっとも有効な方法は，地獄に行く道を熟知すること

皮肉であり逆説的ではありますが，多くの場面に適用できるのではないでしょうか．また，交通事故を起こさないために安全運転を心がけましょう，と呼びかけるよりも，悲惨な交通事故の現場や遺体を見せたほうがよっぽど効き目があるのはなぜでしょうか．多くの人は，つい怠惰であったり，わかってはいるものの波風を立てたくないので，いけないと知りつつアンチパターンにはまっていることもあるでしょう．あるいは，それがアンチパターンであることを知らなかったり，そもそも成功パターンだと信じていて，まさかそれがアンチパターンと気づかない場合もあります．アンチパターンを学習することによる効能は，次のようなものがあります．

- アンチパターンは「問題状況」から「解決策」への橋渡しをする

アンチパターンが起きているかどうかを認識したり，対策を提供するためのテンプレート(型枠)を提供します．これはデザインパターンで陥りがちな錯誤，つまりパターンの適用対象を間違えないためです．

じつは筆者が個人的にアンチパターンを高く評価するのは，ここです．単に「これはダメだ」，「あれはダメだ」とあげつらっているものは，アンチパターンたりえません．ダメな状況を分類し，どう対策するかが系統だてて整理されておらず，単なる文句に成り下がってしまう危険性があるからです．分類され系統だてて整理されることで，どのようにパターンを適用できるか見通しを良くしパターンを使いやすくします．この点において，アンチパターンはデザインパターンの正当な家族(？)といえるでしょう．

- アンチパターンは業界で頻繁に起きている問題と，それらの具体的な解決策を，実体験にもとづき開示する

これは説明するまでもないでしょう．アンチパターンは厳密な意味で学術的とは呼べないかもしれませんが，実践的であるのはたしかです．

- アンチパターンは問題を認識し対策を議論するための共通語を提供する

これはデザインパターンとも共通しますが，ある現象や問題を長文で記述するのではなく短い一言，すなわち共通語を決めることで，議論やコミュニケーションを円滑にできます．たとえばデザインパターンでいえば，「たくさんクラスやモジュールを並べているけど外界と交信させるのは1箇所のクラスにさせる方法だよ．ほら，わかるだろう．キミも少しはソフトウェア開発を経験しているだろうから．え？　大規模プログラミングのことはわからない？　つまり関数がたくさんあるとどれを使っていいか困るから一つの関数だけですませる工夫みたいなものだよ」などと長々と会話するより，「Facadeパターン」の一言で片付くようなものです．

- アンチパターンは組織の資源を多様なレベルで活用することによる総合的な問題解決を支援する

ソフトウェア開発で生じる問題は，その多くは管理の欠陥と組織の欠陥が原因です．ということは，技術要員のみならず，それ以外の要員の協力も必要です．そのため組織全体やあらゆるレベルを視野に納めた記述を心がけることです．

- アンチパターンは業界が頻繁に陥る落とし穴の悲惨さを対象化することでストレスに対するカタルシスをめざす

じつはこのあたりは，筆者が個人的に気にいらない点です．

注1：原文はstovepipe systems．煙突をつぎはぎしながら延長したり故障箇所にツギを当ててボロボロになった状態にたとえている．日本人流にいえば「老舗の温泉旅館」状態というと，わかりやすいかもしれない．

それではサラリーマンが居酒屋で会社や上司の悪口を言っているのと変わらないではないかと考えてしまうからです．しかし，悲惨なことは自分だけではなく業界全体にあるのだと気づき，事態の深刻さがどうしようもないときは巻き添えにならないうちに逃亡するための指針になる点には筆者も同意します．

▶三つの視点からのアンチパターン

アンチパターンは多彩な項目に分かれますが，どの視点（開発者，設計者，管理者のいずれか）から見た場合に成り立つのかで，次の3分類があります．

- 開発の次元のアンチパターン：問題の中心が技術レベルで，おもにプログラマから生まれる
- アーキテクチャの次元のアンチパターン：システムの構造や構成に関わる諸問題
- 管理の次元のアンチパターン：ソフトウェア開発/導入の工程管理や開発組織に関連する諸問題

● 「第２章：アンチパターンの基本形」

デザインパターンとアンチパターンは互いに関連があります（**図1**）．パターンの本質は「問題」とその「解法」です．パターンとは，実践の中に頻繁に観察される「よく見かける解法テクニック」です．

ソフトウェアに関するノウハウの記述とデザインパターンの違いとして，後者はテンプレートを利用する点があげられます．テンプレートは，パターンを構成する解法，設計方法，結果，効能などを形式的に文書化したものです．テンプレートはどのパターンにも採用され，テンプレートを見ることで，パターン採用の可否や，そのパターンを採用した場合の結果などを評価できます．デザインパターンは「問題」と「解法」から成り立ちますが，アンチパターンには，

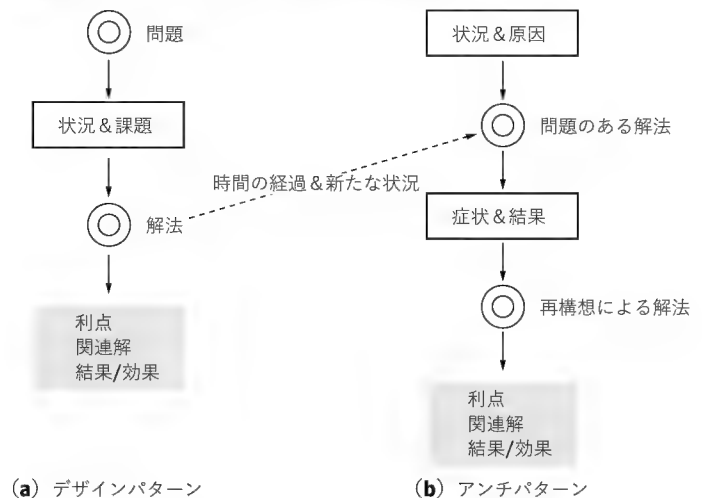
- 「問題のある解法（problematic solution）」：一般的にいたるところで見かけるが，深刻な否定的結果をもたらす解法
- 「再構想による解法（refactored solution）」：アンチパターンを解決してより良い形式に組み直すための一般性のある解法

の二つの解法があります．前述した三つの視点（開発，アーキテクチャ，管理）すべてに共通する用語のベースとして，アンチパターンの基本形を用います．基本形は，アンチパターンの中心的な概念を導く三つの事項を基盤として組み立てられます．三つの事項とは，

- 根本原因：アンチパターンの基盤となる根本状況
- 中心圧力：意思決定の方向と内容を決める中心的な動機
- ソフトウェアデザインの対象レベルの段階：設計の対象レベル，対象視野（システムかアプリケーションかなど）

です．

▶根本原因

根本原因は，ソフトウェア開発で犯される錯誤を指します．錯誤によりプロジェクトは失敗したり予算オーバしたり納期が守れなかったり，ビジネスニーズを満足させられなくなります．根本原因には次の「七つの大罪」があります〔このくだりは聖書に出てくる「七つの大罪」すなわち嫉妬（Envy），暴食（Gluttony），色欲（Lust），怠惰（Sloth），貪欲（Greed），憤怒（Wrath），傲慢（Pride）のパロディだと思われる〕．

〔図1〕デザインパターンとアンチパターンの違い

（**a**）デザインパターン　（**b**）アンチパターン

① 拙速（Haste）：厳しいスケジュールをつきつけられ，品質よりも納期を守れとなってしまう状況

設計がおろそかになされたり，本来はもっと時間をかけて検討や設計をすべきなのに，そうならない状況です．オブジェクト指向アーキテクチャだからといって状況は良くなりませんし，むしろ悪化します．なぜなら品質を保つためには細心の研究（「最新」ではなく「細心」であるのに注意）と，しっかりした意思決定や実験が必要だからです．

② 無関心（Apathy）：正しい解決方法を頭から無視する状況

③ 狭量（Narrow mindedness）：効果的であることが知られている実用性のある解法を拒絶する状況

④ 無精（Sloth）：もっとも最適な方法ではなく，もっとも「安直な」方法を選択したがる状況

⑤ 強欲（Avarice）：必要以上に欲張った作業により必要以上に複雑な事態に追い込まれる状況

たとえば，欲張った設計をすることで必要以上に細部が詳しすぎて抽象化がうまくなされず過剰な複雑性に悩まされることになります．設計者が「複雑なことは賢いことの証明だ」と勘違いしていると，起きやすい状況です．

⑥ 無知（Ignorance）：あらかじめ知っておけば防げたであろうトラブルをわざわざまねいてしまうような状況

⑦ 高慢（Pride）：何もかも自分で作らねば気がすまず，自分が作っていないものを信用しない状況

いわゆる「車輪の再発明」と揶揄される症状は，これや無関心，狭量，無精，無知のいずれかか，それらの組み合わせで起きるのだろうと筆者は思います．

● 「第３章：パターンとアンチパターンのテンプレート」

テンプレートが重要なのは，もしもテンプレートがないと平板な記述となり，あとからどのパターンを適用すべきか検索あ

るいは検討するのに困るからです．それだけならまだしも，はたしてこれはパターンを述べているのか技術説明なのかわかりにくくなったり，「とりとめのない文章」や「主観をダラダラ記述しているだけ」になることでパターンを収集，分類する行為そのものを否定することになり，よりまずい状況になります．それを防止する目的で，パターンを記述するための“記述形式”が望まれ，これがテンプレートになるわけです．

アンチパターンを記述するテンプレートはデザインパターンのものを参考にしますが，第2章で述べられているとおり，二つの解法すなわち「問題のある解法」と「再構想による解法」の両方を記述する必要があります．

▶アンチパターンもどきのテンプレート

これは初期のアンチパターンのテンプレート（と呼ぶにはお粗末な代物）で，

- 名前：そのアンチパターンの呼び名
- 問題：問題点を非難したり，あげつらっているだけ

といったものです．エンターテイメントや単なるうさばらしには役に立つかもしれませんが，実用的ではありません．

▶ミニアンチパターンのテンプレート

アンチパターンの名称と二つの解法を記述したシンプルなテンプレートです．

- 名前：そのアンチパターンの呼び名
- アンチパターンの問題点：どんな方法が繰り返し適用され，そして否定的な結果になったのか
- 再構想による解決：問題点をいかに避け，少なくするか，あるいは再構想するか

▶完全なアンチパターンのテンプレート

詳細な記述をしたテンプレートです．選択項目と書かれている項目は，記述すべき内容がない場合は省略可能です．

- アンチパターンの名前：そのアンチパターンの呼び名
- 別名：そのアンチパターンのほかによく知られた呼び名
- 頻出スケール：そのアンチパターンがどのソフトウェア設計のレベルで頻出するか
- 再構想解の名前：再構想による解に導くパターンの名前
- 再構想解のタイプ：解から帰結する挙動をキーワードにしたもの．ソフトウェア，技術，プロセス，役割のいずれか
- 根本原因：そのアンチパターンの一般的な原因．「七つの大罪」のいずれか
- 対応不全の圧力：そのアンチパターンで無視，誤用，または濫用されている中心圧力
- 挿話証拠：選択項目で，そのアンチパターンに関連して口にされやすいフレーズや喜劇的な素材
- 背景：選択項目で，問題が起きる状況などの有益または興味深い背景的情報
- 一般形式：そのアンチパターンの一般的な性格の図解ないしは一般的な表現
- 症状と結果：そのアンチパターンの症状と結果の箇条書きリスト

Column 2

銀の弾丸

ソフトウェア開発の文献を探すと，「銀の弾丸」や「銀の弾丸などない」という文章にあたることがあります．これはフレッド・ブルックスという人が当時たずさわっていたIBM社のOS/360の開発経験をもとに1986年に発表した論文にあった文章です．銀の弾丸は狼男や吸血鬼を瞬時に撃退するための必須アイテムで，これさえあれば怪物退治が簡単にできることから，ソフトウェア開発を瞬時に解決できる策を「銀の弾丸」にたとえたわけです．

しかし，ホラー映画や怪奇小説の中で銀の弾丸が怪物を倒して人々を救うことはあっても，現実の世界ではソフトウェア開発という怪物を簡単に倒すアイテムなどありえません．にもかかわらず，ありえないアイテムを数々のデベロッパや管理職が求め，そして裏切られてきたわけです．とくに管理職は現場の分析が正確にできているとはいいがたく，銀の弾丸以前に普通の弾丸すら現場にはろくな在庫がない状況だと認識できていません．

たいていは技術的問題ではなく“政治的”問題で，たとえば会社の経営を成り立たせるために無理な値段と無理なスケジュールで仕事を引き受けたからとか，現場の技術レベルに適合しない難しいレベルの仕事とわかりつつ無理に引き受けたからとか，どちらかといえば管理職レベルあるいは経営者レベルで解決すべきだったものを現場に押し付けたのが原因です．となると，いくら現場の技術レベルをあげても解決するはずがなく，かりに技術レベルが上がっても，さらに無理な値段と無理なスケジュールで仕事を受けたり，より難しいレベルの仕事で苦しむことになります．

つまり技術レベルを向上するのと同時に，開発支援環境を向上させないことには問題は解決しないと筆者は考えます．しかし，いくらそのことを指摘しても「わかってはいるんだけど」という力のない反論を聞くのが空しい現状ですが．

ところでブルックスは，『人月の神話（筆者がもっているのは記念増訂版 ISBN4-7952-9675-8 アジソン・ウェスレイ・パブリッシャーズ・ジャパンだが新装版 ISBN4-89471-665-8 ピアソンが出ている）』という開発者の間で長く読み継がれている名著を書いています．ソフトウェア開発にたずさわる人はもちろんのこと管理職や経営者にこそ読んでほしい一冊だと思います．かなり重要な書籍なので，本連載でも機会があれば取り上げるかもしれません．ちなみに有名な「ブルックスの法則」は，同書が出典です．この法則は，

- 遅れているソフトウェアプロジェクトへの要員追加はさらに遅らせるだけだ

という，ブラックユーモアめかしているものの真実だと痛感させるものです．

- 典型的な原因：先に指摘した根本原因に加え，そのアンチパターン独自の原因を列挙
- 例外的なケース：アンチパターンとして指摘したふるまいや工程が問題に結びつかない例外的なケースの指摘
- 再構想による解決：再構想解の説明
- 変種：選択項目で，そのアンチパターンの変種を列挙
- 例：解を適用させる方法を細かく述べる
- 関連対策（や関連アンチパターン）：適切と思われる関連記事を集める
- その他の視点やスケールへの適用性：そのアンチパターンがその他の視点に及ぼす影響を定義する

- 「第４章：アンチパターンの使用上の注意」

アンチパターンを利用して組織のやり方を変えるのは危険をともないます．実際にアンチパターンを組織内で発見し指摘したところ，反社会的人物としてクビになったり左遷された人もいたそうです．組織内に欠陥を見つけて指摘することは個人の一時的な満足になりますが，それはアンチパターンの有意義な利用方法ではありません．どんな組織も複数のアンチパターンを抱えている可能性がありますが，それでも成功している例があります．また，アンチパターンが見つからないことが組織の成功を保証しないことにも注意しましょう．

アンチパターンの目的は，ソフトウェア開発のまずいやり方に焦点を当てることではなく，問題解決のための戦略の立案と実施にあります．

おわりに

『アンチパターン――ソフトウェア危篤患者の救出』の始まりの部分をざっと紹介しましたが，いかがだったでしょうか．第1章から第4章までは同書の第1部「アンチパターン入門」に含まれる部分です．おもに，アンチパターンとは何か，アンチパターンを学習するメリットは何か，アンチパターンの基本形やテンプレートは何かを説明しています．第2部「アンチパターン本論」は実際のアンチパターンにはどのようなものがあるかを詳細に論じます．次回は，第2部を紹介したいと思います．

みやさか・でんと miyadent@anet.ne.jp

回路図形式で演算を行えるツール「TrySignal」の概要

山下伸悟

信号処理ブレッドボードソフトウェア「TrySignal」開発の経緯

信号処理ブレッドボードソフトウェア「TrySignal」は，時系列の数値計算を回路図形式でプログラミングし，その演算を実行できるアプリケーションソフトウェアです．

TrySignalの元になったソフトウェアは，適応型ディジタルフィルタの動作確認のために筆者が作成しました．9年ほど前，筆者はある会社でディジタル変復調システムの開発にソフトウェア担当として参加していました．開発の初期段階ではハードウェアの仕様も固まっていなかったので筆者はいくらか暇だったのですが，そのときプロジェクトリーダから，「復調器の等化フィルタ（適応型フィルタの一種）に関して勉強し，実装方法を検討してほしい」という指示がありました（関連する英語の論文を手渡された）．

勉強もたいへんでしたが，実装についても何らかの方法で動作確認する必要があり，それをどうやってやるかで悩みました．Cのプログラムで作成して確認する方法もありますが，方式やパラメータを変えるたびにプログラムを修正・コンパイル・デバッグするのも面倒です．そこで，シミュレータを作ってみようと考えたわけです．

もう一つ，当時普及しはじめたC++の勉強にちょうど良い題材だと思ったのも大きな理由です．C++では，定義したクラスから複数のインスタンスを生成できます．基本的な演算を行う機能単位（デバイス）をクラスとして定義し，回路に同じデバイスが複数配置された場合は，その数だけインスタンスを生成すればいいわけです．そして「継承」と「ポリモーフィズム」も非常に重要です．これらの機能によって，異なる演算を行う何種類ものデバイスを統一的に扱えるようになります．実際，シミュレータの基本的な部分は2週間ほどで作成できました．

当時はMS-DOS上で動作するもので，テキストファイルで回路を記述していました．用意されているデバイスは25種類で階層化の機能もなく，回路にデバイスを200個程度入れるとメモリが足りなくなってしまいます．それでも，当初の目的のためには十分に役立ちました．筆者としては，役割は果たしたものの，これでおしまいにするのはもったいない，と感じていました．それから，Windowsが普及しプログラミングも身に付けた頃に，GUIで回路図を作成して実行できるシミュレータを作ろうと思い立ち，あるとき会社に企画提案書を提出し，その企画が承認されたのです．

ところが，承認されたはずの企画はさっぱり進行しないのでした．なぜなら，上司が別の仕事を入れてくるのです．これではシミュレータの開発ができないと文句をいうと，「それは優先順位が低いから」と言われてしまいました．このままでは埒があかないと思った筆者は，会社を辞めて個人で開発する決意を固めました．社長にその旨申し入れ，開発権や作成済みの資料などを譲り受ける契約を交わして独立しました．

思い立ってからずいぶん時間がかかりましたが，やっと形にすることができました．最初は「信号処理シミュレータ」と呼んでいましたが，できあがってみると何かをシミュレートするというより，汎用の数値計算ソフトウェアという感じです．そこで「シミュレータ」という呼び方はやめ，「ブレッドボード」と呼ぶことにしました（本誌の読者諸氏には説明不要と思うが，ブレッドボードとは電子回路の試作や実験に使用される基板のこと）．

製品としては，理工系の学生の学習用，あるいは研究，開発部門の方のためのアイデア検証ツールという位置づけです．

「TrySignal」とは？

TrySignalは，Windows上で動作するアプリケーションソフトウェアです．**表1**に動作環境を示します．**図1**は，作成した回路の演算を実行した画面の例です．一見，回路シミュレータに似ていますが，TrySignalにおける回路アイテムは，実際の電子部品をシミュレートするものではありません．四則演算や数学関数などの計算機能を単位として，その間でデータをやり取りできるようにしたものです（もちろん，シミュレータとして使用することもモデルしだいでは可能）．TrySignalはシェアウェアとして公開されており，`http://www.digi-prove.com/`からダウンロードして試用できます．

〔表1〕動作環境

対応OS	Microsoft Windows 95/98/98SE/Me/NT4.0/2000/XP（日本語版のみ）
CPU	Intel Pentium およびその互換 CPU
メモリ容量	OS が起動した状態での空きメモリ 6M バイト以上（ただし，作成する回路の規模により増加）
インストールに必要なハードディスク容量	約 3M バイト
画面の解像度	640 × 480 以上（1024 × 768 以上を推奨）
画面の色数	16 色以上（256 色以上を推奨）
ポインティングデバイス	マウスまたは同等のデバイスが必要（Microsoft 社製ホイール付きマウスを推奨）

デバイスと信号データ

回路を構成するためのアイテムにはいくつかの種類がありますが，中心となるのは「デバイス」と呼ばれるアイテムです．デバイスには，実行ファイルに含まれている「組み込みデバイス」と，DLL ファイルの形で後から追加される「プラグインデバイス」があります．

四則演算など比較的単純な機能のものは，組み込みデバイスとして用意されています．組み込みデバイスは全部で 121 種類あり，「基本発生器」，「乱数発生器」，「基本演算」，「数学関数」，「特殊・工学関数」，「複素数演算」，「統計演算」，「論理・ビット演算」，「時系列演算」，「制御その他」の 10 のカテゴリに分類されています．**図 2** は，デバイスを選択する画面の例です．デバイスのリストを掲載したかったのですが，誌面の都合により割愛します（前述の Web ページにデバイスリストも掲載されているので，アクセス可能な方はそちらをご覧いただきたい）．グラフ表示やファイル入出力などのデバイスは，プラグインデバイスとして用意されています．今後，いろいろな機能をもったプラグインデバイスが追加される予定です．

デバイス間でやりとりされる信号のデータ型には，次のものがあります．ただし，すべてのデバイスがすべてのデータ型に対応するわけではなく，デバイスによってサポートするデータ型は異なります．

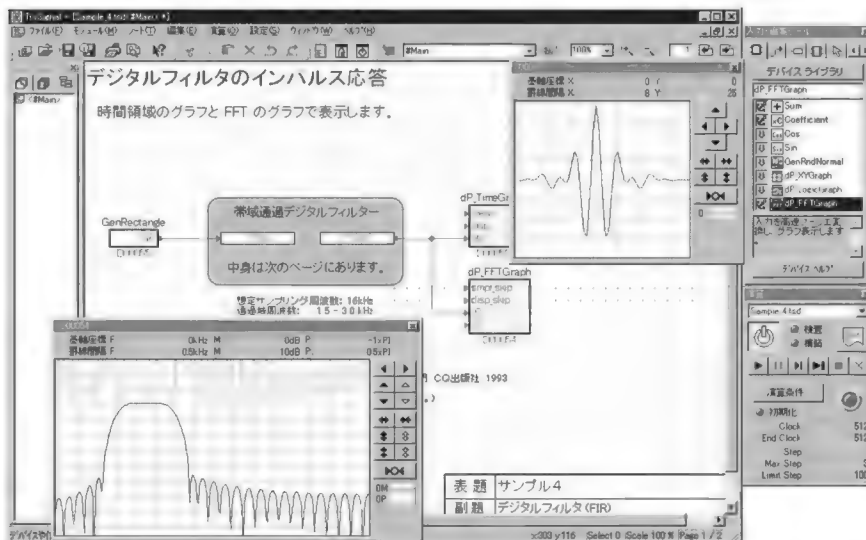
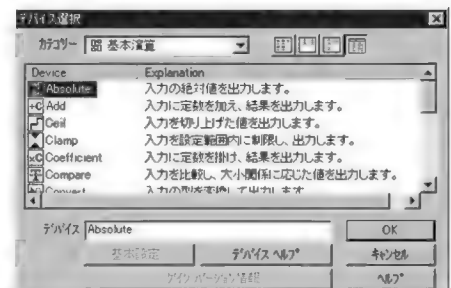
> 32 ビット符号付き整数，32 ビット符号なし整数，64 ビット符号付き整数，64 ビット符号なし整数，64 ビット浮動小数点数，テキスト，拡張型

テキスト型は，たとえばファイルを扱うデバイスで，その入力端子からファイルのパス名を与える場合などに使用されます．拡張型は，将来の拡張用に用意されているものです．回路図上では，回路アイテムの端子や信号線は，信号のデータ型にしたがって色を変えて表示されます．データ型と色の対応は，ユーザーが変更できます．

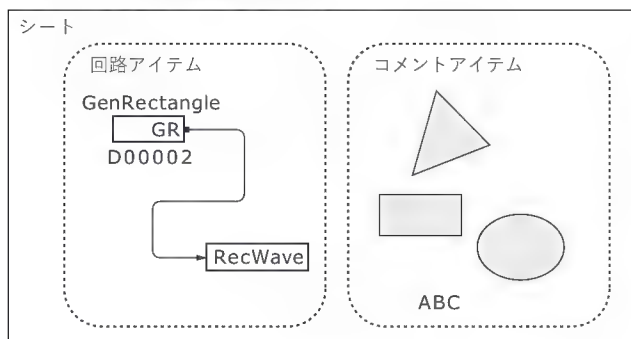
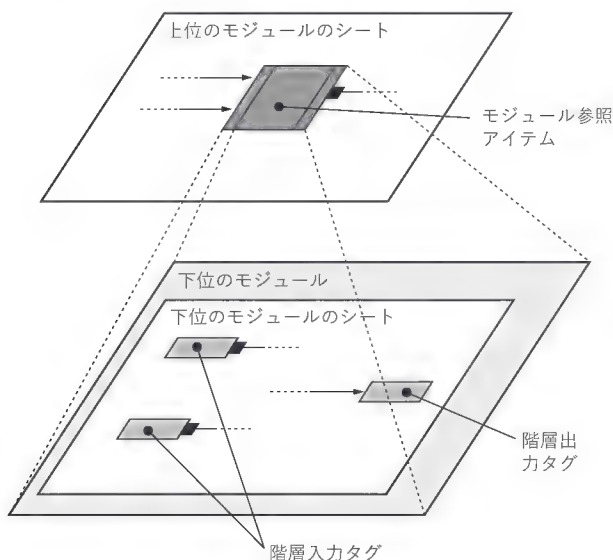
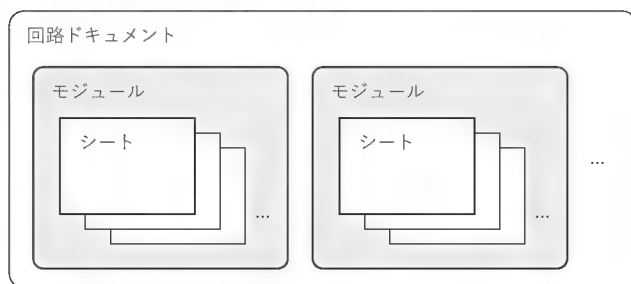
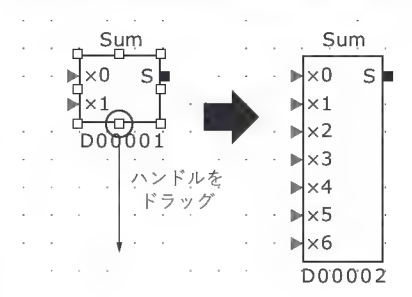
基本的な数値計算を行ういくつかのデバイスは，複数の数値型に対して柔軟に対応します．たとえば，入力された複数の値の合計を出力する Sum というデバイスは，入力として 5 種類の数値型のどれでも受け付け（混在可），プロパティで設定された型で出力します．計算における型変換は，C 言語の式と同様です．ただし，浮動小数点に関してはオーバフローなどのエラーチェックを行っており，エラーが発生した場合は演算を停止します（設定により，一時停止や無視して続行することもできる）．

回路の構成

回路を構成する作業は，ドローソフトで図を作成する感覚で行えます．回路図に説明を付けられるように，回路アイテムのほかに「コメントアイテム」と呼ぶ，テキストや単純な図形のアイテムも用意されています（**図 3**）．どのアイテムを入力するか

〔図1〕TrySignal で演算を実行した画面

〔図2〕デバイスを選択する画面

〔図3〕シート上に配置されるアイテム

〔図4〕
入力・編集用ダイアログ
(矩形入力モード)

〔図5〕サイズ変更による端子数の指定の例

〔図6〕回路の階層化

〔図7〕回路ドキュメントの構成

は，メニューまたは「入力・編集ツール」ダイアログで選択します(**図4**)．端子数が可変のデバイスでは，デバイスを選択したときに表示されるハンドル(小さな矩形)をドラッグしてサイズを変えると，それに応じて端子数が変わるという仕様になっています(**図5**)．

回路は，複数のシートに分割して作成することもできます．シート間の信号の接続は，「宣言タグ」と「参照タグ」と呼ばれるアイテムで行います．宣言タグは信号に名前を付ける機能があり，参照タグは名前の付いた信号を参照する機能があります．もちろん，同一シート内で宣言，参照を行ってもかまいません．

TrySignalでは回路を「モジュール」という単位で作成することにより，階層的に表現できます(**図6**)．単純な機能をもった回路をモジュールとして定義し，それを組み合わせることで規模の大きな回路をわかりやすく構成することができます．

最上位のモジュール(メインモジュール)は必ず「#Main」という名前になります．その他のモジュールには任意の名前を付けることができます．階層の深さは最大6まで可能です．メインモジュールから参照関係でつながっているモジュールが，演算の対象となります．メインモジュール以外でほかから参照されないモジュール(孤立モジュール)も存在できますが，演算の対象にはなりません．

一つまたは複数のモジュールから構成された回路全体をまとめて，「回路ドキュメント」と呼びます(**図7**)．ファイルに保存する場合，一つの回路ドキュメントが一つのファイルになります．モジュールの再利用のために，モジュールの階層グループを単位として，ファイルに保存したり読み込んだりする機能もあります．

演算の実行

回路を構成できたらそれを動作させることができますが，この動作状態のことを「演算モード」と呼んでいます．概念的には，「クロック」と呼ばれる単位で回路が同期的に動作します．クロックとは時間の刻みを意味し，(モデル上は)一定の間隔をもっています．デバイスの演算スピードはどのデバイスも同じで，クロック周期より充分小さいと仮定されています．デバイスの出力端子から入力端子に伝達される信号も，クロック周期より充分小さい，一定の遅延をもつと考えます．

一つのデバイスは，一つのクロックで1回だけ計算を行う，

というわけではありません．クロック内で何回か計算が繰り返されることがあり，この計算の繰り返しの単位を「演算ステップ」と呼びます．クロック内でデバイスと信号の状態が収束するまで，演算ステップが繰り返されます．

演算の内部的な動作についてはもう少し詳しく説明したいところですが，長くなるので省略します．TrySignalのヘルプで詳しく説明しているので，興味のある方はそちらをご覧ください．演算モードでは，演算を連続的に実行するだけでなく，途中で一時停止させたり，クロック単位あるいは演算ステップ単位で実行できます(**図8**)．

〔図8〕演算モードでの操作を行うダイアログ

〔図9〕dP_XYGraphの表示例

一時停止状態では，個々の信号の値を確認することもできます．内部的にエラーが発生した場合はログファイルに記録されるので，それを確認することで問題が発生したデバイスやタイミングがわかります．

演算の実行内容を確認するには，いくつか用意されているグラフ表示デバイスを使用するのが便利です(**図9**)．これらは演算の実行中でも上下や左右に拡大，縮小，シフト表示が可能です．ウィンドウ自体も，サイズ変更や表示/非表示の切り替えができます．ただし，これらのデバイスはあくまで確認用であり，印刷機能もありません．きれいなグラフを作成したい場合は，dP_CSVWriteデバイスでCSV形式ファイルにデータを書き出し，グラフ作成機能をもつ他のアプリケーションソフトを使用するのが良いでしょう．CSV形式のファイルは，表計算ソフトなどでも読み込むことができます[注1]．

追加のプラグインデバイス

2003年7月現在，「WAVファイル入出力プラグイン」，「BMPファイル入出力プラグイン」，「手動調節器プラグイン」が別途ダウンロード可能になっています．WAVファイル入出力プラグインは，音声データ用ファイルであるWAV形式ファイルを扱えます．読み出した音声データにエコーやサラウンド効果を加え，別のファイルに書き出す，といったことが可能になります．5.1chサラウンドなど，多チャネルWAVファイルにも対応しています．Webページには，これを使用して作成したWAVファイルのサンプルもあるので，アクセス可能な方は聞いてみてください．

BMPファイル入出力プラグインは，画像用ファイルであるBMP形式ファイルを扱うことができます．カラー変換や近傍フィルタ処理(ぼかしやエンボス効果など)を行えます．これも，サンプル画像がWebページに置いてあります．手動調節器プラグインには，演算の実行中に手動でパラメータを変えたい場合に便利なデバイスが4種類含まれています．

おわりに

近年，子供たちの理科離れ，数学離れが進み，学力も低下しているといわれています．「技術」とはすでに存在し(あるいは誰か知らない人が開発し)，製品を利用すればよいだけのものであって，自ら新しいものを作るなど思いもよらない――多くの若者はそのように感じているのではと思います(若者に限らない？)．

何とか若い人の興味を引くような教育を行う必要があるわけですが，その一つとして，パソコンを使った教育も行われています．ところが，学校や研究所のパソコンには高価なソフトウェアがインストールされていて使えるけれども，個人のパソコンではそれが使えない，という状況があります．もちろん，そういったソフトウェアは高機能であり，それに見合った価格だとは思いますが，個人で買って気軽に使えるソフトウェアも必要なのではないでしょうか？

TrySignalを開発しようと思った動機の一つは，「個人で買える学習用ソフトウェア」を作るということでもありました(ちなみに，TrySignalはゲームソフトを2～3本我慢すれば買える価格に設定した)．TrySignalについては，ここで説明しきれなかった機能もたくさんあるので，ぜひダウンロードしてお試しいただければと思います．そして，このソフトウェアが皆様のお役に立てば幸いです．

やました・しんご　デジプルーブ　http://www.digi-prove.com/

注1：TrySignalには，CSV形式ファイルを読み込むためのdP_CSVReadというデバイスも用意されており，これとdP_CSVWriteを組み合わせると，表計算ソフトで行うような集計処理も可能になる．もっとも，表計算ソフトに比べると個々のデータの視認性は悪いが．

第12回 続・GCC2.95から追加変更のあったオプションの補足と検証

岸　哲夫

今回は前回(2003年8月号)に引き続き，GCC2.95から追加変更のあったオプションの補足と検証を行う．その前に，蛇足かもしれないがGCC3.3を導入する方法について解説する．
(筆者)

GCC3.3をインストールする方法

筆者の環境はRed Hat Linux 8.0でした．その場合，glibcのバージョンを上げないと不具合が生じるとの情報がGNUのサイトにあったので，Red Hat Linux 9.0に変更しました．「Build status for GCC 3.3」には他のディストリビューションの情報もあるので，参考になるでしょう．URLは以下のとおりです．

http://gcc.gnu.org/gcc-3.3/buildstat.html

では，具体的なインストール方法を解説します．

1) http://www.dnsbalance.ring.gr.jp/のRingServerなどで最新版のソースをダウンロードします．

2) bzcat gcc-3.3.tar.bz2 | tar xvf-
上記のコマンドで展開を行います．

3) cd gcc-3.3で移動．

4) 以下のようにconfigureを行います．

```
--prefix=/usr/local/gccbinutils
        --enable-__cxa_atexit --enable-shared
                    --enable-threads=posix
```

これは，環境や用途にも依存します．

筆者はGCC3.2.2も比較のため残しておきたかったので，/usr/local/gccbinutils/bin/gccにバイナリを作成することにします．

5) 以下のようにmakeコマンドを入力します．

```
make CFLAGS='-O2' LIBCFLAGS='-g-O2'
              LIBCXXFLAGS='-g-O2-fno-implicit
                    -templates' bootstrap
```

同じく環境や用途に依存しますが，Pentium III，800MHzのマシンで2時間少しコンパイル時間がかかります．

6) make installでインストールします．

7) mv /usr/bin/gcc /usr/bin/gcc322

```
ln -s /usr/local/gcc binutils/bin/gcc/usr/
                                   bin/gcc
```

これで旧版gccをgcc322に変更し，新版(3.3)gccをgccと打鍵して使えるようにしました．

8) gcc -vで以下のようになるはずです．

もちろんCPUなどの環境によって少し違うはずです．

```
$gcc -v
/usr/local/gccbinutils/lib/gcc-lib/i686-pc-
        linux-gnu/3.3/specsからspecを読み込み中
コンフィグオプション: ./configure --prefix=/usr/
    local/gcc binutils --enable-__cxa_atexit
      --enable-shared --enable-threads=posix
スレッドモデル : posix
gccバージョン3.3
```

C言語の方言を扱うオプションの補足

前回で書ききれなかった，GCC2.95から追加変更のあったオプションの補足と検証を行います．

● -ffreestanding

このオプションは組み込み環境や，カーネルをコンパイルする際に使用します．標準ライブラリが存在しない環境や，プログラムのスタートが__mainではない環境において使用します．

これを指定すると同時に-fno-builtinオプションを指定したことになります．連載第3回で-fno-builtinオプションの解説をしましたが，GCC3.3になって，その意味が変わっています．これも後述します．

● -fms-extensions

マイクロソフト製のヘッダファイル中にある非標準的記述を受け入れます．これはCygwin環境で使用する際に，VC++などで作った環境を移植する際に使用します．

● -ansi

前回の連載の補足です．GCC2.95では-ansiオプションを使用した場合，関数alloca/abort/exit/_exitは組み込み関数として扱われません．

GCC3.3で-ansiオプションを使用した場合，ANSI標準でないallocaのような関数は組み込み関数として扱われません．組み込み関数については別に項を設け，そこで詳しく解説します．

● -fno-asm

従来はキーワードinlineはANSI標準キーワードでなかったのですが，ANSI C99規格からは標準となりました．よってこのオプションを指定すると，asm，typeofをキーワードとして認識しないようになります．これらの名前は識別子として使用可能になり，代わりに__asm__，__typeof__がキーワードとして使えるようになります．

● -fallow-single-precision

連載第3回で解説したオプションですが，GCC3.3では廃止されました．

以上のC言語の方言を扱うオプションを**表1**にまとめます．

〔表1〕C言語の方言を扱うオプション

バージョン2.95	バージョン3.3	
-ansi	-ansi	
	-aux-info filename	追加
-fallow-single-precision		廃止
-fcond-mismatch	-fcond-mismatch	
-ffreestanding	-ffreestanding	
-fhosted	-fhosted	
	-fms-extensions	追加
-fno-asm	-fno-asm	
-fno-builtin	-fno-builtin	
	-fno-builtin-「function」	追加
-fno-signed-bitfields	-fno-signed-bitfields	
-fno-unsigned-bitfields	-fno-unsigned-bitfields	
-fsigned-bitfields	-fsigned-bitfields	
-fsigned-char	-fsigned-char	
-fstd=		廃止
-funsigned-bitfields	-funsigned-bitfields	
-funsigned-char	-funsigned-char	
-fwritable-strings	-fwritable-strings	
	-no-integrated-cpp	追加
	-std=	追加
-traditional	-traditional	
-traditional-cpp	-traditional-cpp	
-trigraphs	-trigraphs	

LINK関連のオプションの補足

新しいオプション-l libraryは-llibraryと同義です．

● -nodefaultlibs

このオプションを指定すると，リンク時に標準のシステムライブラリを使用しません．

● -shared-libgcc

共有ライブラリとしてlibgccを供給するシステムにおいて，このオプションは，共有されたライブラリの使用を強制します．

もっともコンパイラを構築したときに，libgccの共有されたバージョンが造られなかったなら，このオプションは，効果を持ちません．

● -static-libgcc

このオプションは上と逆に静的ライブラリとしてリンクすることを強制します．

以上を**表2**にまとめます．

〔表2〕LINK関連のオプション

バージョン2.95	バージョン3.3	
-c	-c	
-E	-E	
-llibrary	-l library	
	-llibrary	追加
-lobjc	-lobjc	
-nodefaultlibs	-nodefaultlibs	
-nostartfiles	-nostartfiles	
-nostdlib	-nostdlib	
'object-file-name	'object-file-name	
-S	-S	
-s	-s	
-shared	-shared	
	-shared-libgcc	追加
-static	-static	
	-static-libgcc	追加
-symbolic	-symbolic	
-u symbol	-u symbol	
"-Wl,option"	"-Wl,option"	
-Xlinker option	-Xlinker option	

プリプロセッサ関連のオプションの補足

● -Wunused-macros

このオプションを使うと，ソース中に使用していないマクロがあった場合に警告します(**リスト1**)．

コンパイルの結果を以下に示します．

```
$gcc -Wunused-macros test165.c
test165.c:5:1: 警告: macro"max"is not used
$gcc test165.c
$
```

〔リスト1〕マクロが未使用の場合警告する例(test165.c)

```
/*
 *使用していないマクロに警告する
 */
#include <stdio.h>
#define max(a,b) ((a) >? (b))
int main(void)
{
    long    b;
    printf("test165¥n");
    return  0;
}
```

● -Wendif-labels

このオプションを使うと，プリプロセッサ文中の意味のないテキストに警告します(**リスト2**)．

コンパイルの結果を次に示します．

```
$gcc-Wendif-labels test166.c
test166.c:7:7: 警告:
余分なトークンが #else ディレクティブの終りにあります
test166.c:9:8: 警告:
余分なトークンが #endif ディレクティブの終りにあります
$
```

なお，このオプションはデフォルトで使われます．

```
$gcc test166.c
test166.c:7:7: 警告:
余分なトークンが #else ディレクティブの終りにあります
test166.c:9:8: 警告:
余分なトークンが #endif ディレクティブの終りにあります
$
```

- -CC

マクロを展開する際に，コメントをマクロ展開リストに出力するか否か指定します(**リスト3**)．

コンパイル(マクロ展開)の結果を以下に示します．

```
$gcc -CC -E test167.c|grep [ab]1
  long a1=1000; //コメント
  long b1=1000; /*test*/
$gcc -E test167.c|grep [ab]1
  long a1=1000;
  long b1=1000;
$
```

オプション-Eは，前処理のリストを標準出力に出力するものです．-CC単独では何もしません．

- -###

-vオプションと同様にバージョンを表示します．

- -x language

いくつかの言語を混在させてコンパイルを行う際に，次の-xオプションまでlanguageとして扱うオプションです．しかし*.cや*.sと接尾語でそのソースが何かを確定させるほうが効率的です．

- -x none

このオプションを指定すると，-x languageで指定した言語を無視し，接尾語で確定した言語として扱います．

- -pass-exit-codes

通常はgccコマンドでコンパイル・リンク中にエラーコード1を戻したところで終了してしまいます．このオプションを指定すると，最後まで実行します．ただしコンパイル・リンクエラーより，もっと重要な異常が起きた場合には終了してしまいます．

- -D マクロ名

このオプションに続いてマクロ名を指定すると，それは1と定義されます(**リスト4**)．

コンパイルと実行の結果を以下に示します．

```
$gcc test174.c -o test174
$ ./test174
βテスト中
$gcc -D dbg test174.c -o test174
$ ./test174
デバッグ中
$
```

2回目のコンパイル・実行ではdbgが1と定義されたため“デバッグ中”と出力するコードをコンパイルしました．

〔リスト2〕プリプロセッサ文中の意味のないテキストに警告する例(test166.c)

```
/*
 *else,endif の後のテキストに警告する
 */
#include <stdio.h>
#if TEST166
    long    a1  =0;
#else TEST166
    long    a1  =1000;
#endif TEST166
#if TEST166a
    long    a2  =0;
#else
    long    a2  =1000;
#endif
int main(void)
{
    long    b;
    printf("test166¥n");
    return  0;
}
```

〔リスト3〕マクロ展開リスト中のコメントを有効にする例(test167.c)

```
/*
 *マクロ中のコメントを有効にする
 */
#include <stdio.h>
#if TEST167
    long    a1  =0; //test
#else
    long    a1  =1000;  //コメント
    long    b1  =1000;  /*test*/
#endif
int main(void)
{
    long    b;
    printf("test167¥n");
    return  0;
}
```

〔リスト4〕-Dオプションの例(test174.c)

```
/*
 *-Dオプションの例
 */
#include <stdio.h>
int main(void)
{
#if dbg
    printf("デバッグ中¥n");
#else
    printf("βテスト中¥n");
#endif
    return  0;
}
```

〔リスト5〕**-Dオプションでマクロに値を設定する例**(test175.c)

```
/*
 *Uオプションでマクロを設定する例
 */
#include <stdio.h>
int main(void)
{
    printf("macro_01の値は%d¥n",macro_01);
    printf("macro_02の値は%d¥n",macro_02);
    return  0;
}
```

- -D マクロ名＝値

このオプションでマクロに値を設定できます(**リスト5**).

コンパイルと実行の結果を以下に示します.

```
$gcc -D macro_01="58548" -D macro_02
                    ="152635" test175.c -o test175
$ ./test175
macro_01の値は58548
macro_02の値は152635
$
```

- -U マクロ名

通常は__GNUC__という名前がspecsによって初期設定されています. これはGCCのメジャーバージョンなので, この環境では3です. これを-U __GNUC__とオプションで指定した場合, __GNUC__の定義が無効になります(**リスト6**).

コンパイルと実行の結果を以下に示します.

```
$gcc -U __GNUC__ test176.c -o test176
$ ./test176
__GNUC__は無効
$gcc test176.c -o test176
$ ./test176
__GNUC__は有効
$
```

- -A predicate=answer

このオプションはプログラム中の条件に値を与えることができます. 条件コンパイルなどに使用することができます(**リスト7**).

コンパイルと実行の結果を以下に示します.

```
$gcc -A predicate=test test178.c -o test177
$ ./test177
predicate test
$gcc test177.c -o test177
$ ./test177
predicate answerは定義されていません
$
```

- -A-predicate = answer

上で説明したオプションの逆の意味をもちます. プログラム中の条件の値を取り消します. これはspecsファイルなどに初期設定されているものを解除するときなどに使います.

〔リスト6〕**Uオプションでマクロ設定を取り消す例**(test176.c)

```
/*
 *Uオプションでマクロ設定を取り消す例
 */
#include <stdio.h>
int main(void)
{
#if __GNUC__
    printf("__GNUC__は有効¥n");
#else
    printf("__GNUC__は無効¥n");
#endif
    return    0;
}
```

〔リスト7〕**-Aオプションでpredicateとanswerを設定する例**(test177.c)

```
/*
 *-Aオプションでpredicate と answerを設定する例
 */
#include <stdio.h>
int main(void)
{
#if #pre (test)
    printf("predicate test ¥n");
#else
    printf("predicate answerは定義されていません¥n");
#endif
    return  0;
}
```

以上のプリプロセッサ関連(全般にかかわるオプション)のまとめを**表3**に示します.

警告を要求/抑止するオプションの補足

- -Wno-format-zero-length

-Wformatオプションを使用するときに, 同時にこのオプションを指定するとprintfなどのフォーマットの長さが0でも警告しません(**リスト8**). -Wformatに関しては連載第3回で解説しています.

コンパイルの結果を以下に示します.

```
$gcc -Wformat -Wno -format-zero
                          -length test168.c
test168.c: 関数 `main' 内:
test168.c:8: 警告: フォーマットはdoubleですが,
              引数はdifferent typeです(引数2)
$gcc -Wformat test168.c
test168.c: 関数 `main' 内:
test168.c:8: 警告: フォーマットはdoubleですが,
              引数はdifferent typeです(引数2)
test168.c:9: 警告: zero-length printf
                              format string
$gcc test168.c
$
```

〔表3〕プリプロセッサ関連(全般にかかわるオプション)のまとめ

バージョン2.95	バージョン3.3	
	-###	追加
-Aquestion(answer)		廃止
	-A -predicate=answer	追加
	-ansi	追加
-C	-C	
-c	-c	
	-CC	追加
	-D name	追加
	-D name=definition	追加
	-dCHARS	追加
-dD		廃止
-dM		廃止
-Dmacro		廃止
-Dmacro=defn		廃止
-dN		廃止
-E	-E	
	file.adb	追加
	file.ads	追加
file.c	file.c	
file.C	file.C	
	file.c++	追加
file.cc	file.cc	
	file.cp	追加
file.cpp	file.cpp	
file.cxx	file.cxx	
	file.f	追加
	file.F	追加
	file.for	追加
	file.FOR	追加
	file.fpp	追加
	file.FPP	追加
file.h	file.h	
file.i	file.i	
file.ii	file.ii	
file.m	file.m	
	file.mi	追加
	file.r	追加
file.s	file.s	
file.S	file.S	
	-fno-show-column	追加
	-fpreprocessed	追加
	-ftabstop=width	追加
	-gcc	追加
-H	-H	
--help	--help	
	I	追加
	-I-	追加
	-I dir	追加
-idirafter dir	-idirafter dir	
-imacros file	-imacros file	
-include file	-include file	
-iprefix prefix	-iprefix prefix	
-isystem dir	-isystem dir	
-iwithprefix dir	-iwithprefix dir	
-iwithprefixbefore dir	-iwithprefixbefore dir	
-M	-M	
-MD	-MD	
	-MF file	追加
-MG	-MG	
-MM	-MM	
-MMD	-MMD	
	-MP	追加
	-MQ target	追加
	-MT target	追加
	N	追加
-nostdinc	-nostdinc	
	-nostdinc++	追加
-o file	-o file	
other	other	
-P	-P	
	-pass-exit-codes	追加
	-pedantic	追加
	-pedantic-errors	追加
-pipe	-pipe	
	-remap	追加
-S	-S	
	-std=standard	追加
	--target-help	追加
	--target-help	追加
	-traditional-cpp	追加
-trigraphs	-trigraphs	
-Umacro		廃止
	-U name	追加
-undef	-undef	
-v	-v	
	-version	追加
	--version	追加
	-x assembler-with-cpp	追加
	-x c	追加
-x language	-x c++	
-x none	-x language	
	-x none	追加
	-x objective-c	追加

〔リスト8〕**-Wformatオプションの警告を一部抑止する例**(test168.c)

```
/*
 *-Wformatオプションの警告を一部抑止する
 */
#include <stdio.h>
int main(void)
{
    long    b   =   100000;
    printf("%f¥n",b);
    printf("","test168¥n");
    return  0;
}
```

● -Wnonnull

第7回の連載で触れましたが，GCCでは「関数属性の宣言」が可能です．この拡張機能の3.2.2になったことによる追加は後述します．

追加された関数属性の宣言の中に関数の引き数がNULLではまずい場合，エラーメッセージで警告する機能があります(**リスト9**)．

その機能を付加したい場合に，関数属性の宣言をして，この

〔リスト9〕関数の引き数がNULLではまずい場合にエラーメッセージで警告する例(test169.c)

```
/*
 *-Wnonnull 関数の引き数がNULLではまずい場合に
 *エラーメッセージで警告する
 */
#include <stdio.h>
void test_memcpy(void *dest, const void *src, size_t len)
    __attribute__((nonnull (1, 2)));    //関数属性の宣言

int main(void)
{
    void    *a;
    test_memcpy(NULL,NULL,100);
    test_memcpy((void*)strcpy(NULL,NULL),a,100);
    return  0;
}
void test_memcpy(void *dest, const void *src, size_t len)
{
}
```

オプションを使用します．

コンパイルの結果を以下に示します．

```
$gcc -Wnonnull test169.c
test169.c: 関数 `main' 内:
test169.c:12: 警告: null argument where
                    non-null required(arg 1)
test169.c:12: 警告: null argument where
                    non-null required(arg 2)
$
```

ソースを見てわかるように，関数属性の宣言をしても，明白にNULLを指定した場合のみ作用します．“strcpy(NULL,NULL)”の戻り値は確実にNULLだと思いますが，チェックすることはできません．もちろん実行時にNULLになるような式を指定してもチェックできません．

● -Wswitch-default

これは，switch文のdefault動作を指定する文がない場合に警告させるためのオプションです(**リスト10**)．

コンパイルと実行の結果を，以下に示します．

```
$gcc test170.c -o test170
$gcc -Wswitch-default test170.c -o test170
test170.c: 関数 `main' 内:
test170.c:17: 警告: switch missing
                            default case
$ ./test170
びろーん
違います
$
```

● -Wswitch-enum

C言語の型に「列挙型」と呼ばれるものがあります．switch文のインデックスに列挙型の項目すべてが指定されていない場合に，警告させるためのオプションです(**リスト11**)．

コンパイルの結果を次に示します．

〔リスト10〕switch文のデフォルトがない場合に警告する例(test170.c)

```
/*
 *switch文のデフォルトがない場合に警告
 */
#include <stdio.h>
int main(void)
{
    int     num;
    num =   1;
    switch  (num)
    {
        case 1:
            printf("びろーん¥n");
            break;
        case 2:
            printf("がちょーん¥n");
            break;
    }
    num =   9;
    switch  (num)
    {
        case 1:
            printf("びろーん¥n");
            break;
        case 2:
            printf("がちょーん¥n");
            break;
        default:
            printf("違います¥n");
            break;
    }
    return  0;
}
```

〔リスト11〕列挙型の項目が足りない場合に警告する例(test171.c)

```
/*
 *列挙型の項目が足りない場合に警告
 */
#include <stdio.h>
int main(void)
{
    enum tag1 { a , b , c , d , e } ;
    enum tag1 x ;
    switch  (x)
    {
        case a:
            printf("その1¥n");
            break;
        case b:
            printf("その2¥n");
            break;
        case c:
            printf("・・・・・・¥n");
            break;
        case e:
            printf("まだまだ¥n");
            break;
    }
    return  0;
}
```

```
$gcc -Wswitch-enum test171.c -o test171
test171.c: 関数 `main' 内:
test171.c:23: 警告: enumeration value`d'not
                         handled in switch
$
```

このように，列挙型の項目の値をすべてswitch文の処理に対応させたい場合に，指定すると便利です．抜けをチェックできます．

● -Wstrict-aliasing

ANSIでも以前から規定されていますが，あるオブジェクトに格納された値にアクセスする方法は，次のうちいずれか一つの型をもつ左辺値によるものだけでなければなりません．

- オブジェクトの宣言された型
- オブジェクトの宣言された型の修飾版
- オブジェクトの宣言された型と対応する符合付き整数型または符合なし整数型
- メンバの中にこれらの型の一つを含む集成体または共用体型
- 文字型

この規定を破っても警告は出ません．結果が不定になるだけです．

また，あるオブジェクトと別の名前をもつオブジェクトは，違うアドレスに配置されているという前提で，コンパイラは最適化を行います．規則違反の別名定義があると，最適化をすることができなくなります．このオプションを指定すると，別名規則を破っていて最適化に不都合な場合に警告します(**リスト12**)．

コンパイルの結果を以下に示します．

```
$gcc -fstrict-aliasing
                  -Wstrict-aliasing test172.c
test172.c: 関数 `main' 内:
test172.c:10: 警告: dereferencing
                type-punnedpointer will break
                        strict-aliasing rules
test172.c:11: 警告: register変数 `b'
                        のアドレスが要求されました
test172.c:11: 警告: dereferencing
                type-punned pointer will break
                        strict-aliasing rules
$gcc test172.c
test172.c: 関数 `main' 内:
test172.c:11: 警告: register変数 `b'
                        のアドレスが要求されました
$
```

もともと別名定義のできるわけがないレジスタ変数を，別名でアクセスしようとすると，別の警告が出ます．この件に関しては，このオプションを外しても警告を出します．

● -Wno-div-by-zero

このオプションを付けると，明らかなゼロ除算がソース中にあっても警告しないようにします．NANやINFを導出させるのに使用することがあります．連載の第10回でも少し触れましたがNAN，INFはマクロで定義されています．NANは数値例外であり，INFは無限大です(**リスト13**)．

コンパイルの結果を以下に示します．

```
$gcc test173.c -o test173
test173.c: 関数 `main' 内:
test173.c:8: 警告: division by zero
$gcc -Wno-div-by-zero test173.c -o test173
$ ./test173
NAN
INF
$
```

● -W

まぎらわしいですが，これは大文字のWです．このオプションは，雑多な文法ミスに警告を発してくれます(**リスト14**)．

コンパイルの結果を以下に示します．

```
$gcc -W test178.c -o test178
test178.c: 関数 `foo' 内:
test178.c:10: 警告: type of`a'
                        defaults to`int'
$
```

〔**リスト12**〕**別名規則を破っていて最適化に不都合な場合に警告する例**(test172.c)

```
/*
 *別名規則を破っていて最適化に不都合な場合に警告
 */
#include <stdio.h>
int main(void)
{
    int    a        =    100;
    register int    b    =    200;
    double  *x;
    x   =   (double*)&a;
    x   =   (double*)&b;
    printf("%d¥n",a);
    printf("%f¥n",x);
    return 0;
}
```

〔**リスト13**〕**ゼロ除算で警告しないようにする例**(test173.c)

```
/*
 *ゼロ除算で警告しないようにする
 */
#include <stdio.h>
#include <math.h>
int main(void)
{
    float f  = 10.0/0;
    double d = 0.0/0.0;
    printf("%G¥n",f);
    printf("%G¥n",d);
}
```

〔**リスト14**〕**-Wオプションでエラーになる例**(その1)(test178.c)

```
/*
 *-Wオプションでエラーになる例
 */
#include <stdio.h>
int main(void)
{
    return  0;
}
foo (a)
{
 if (a > 0)
   return a;
}
```

〔リスト15〕-Wオプションでエラーになる例(その2)(test179.c)

```
/*
 *-Wオプションでエラーになる例
 */
#include <stdio.h>
int main(void)
{
    char    tbl[50];
    int     i   =   10;
    int     j   =   20;
    tbl[i,j]    =   'a';
    if  (tbl[10]    ==  'a' )
    {
        printf("tbl[10]は'a'¥n");
    }
    if  (tbl[20]    ==  'a' )
    {
        printf("tbl[20]は'a'¥n");
    }
    return  0;
}
```

これは引き数に型を指定しなかったため，int型だと受け取られコンパイルした例です(**リスト15**)．

コンパイルと実行の結果を以下に示します．

```
$gcc -W test179.c -o test179
test179.c: 関数 `main' 内:
test179.c:10: 警告: left-hand operand of
                  comma expression has no effect
$ ./test179
tbl[20]は'a'
$gcc test179.c -o test179
$ ./test179
tbl[20]は'a'
$
```

たとえば，COBOLプログラマが思い込みで二次元テーブルの使い方を間違え，カンマで区切ってしまった場合に警告ができます(**リスト16**)．

コンパイルの結果を以下に示します．

```
$gcc -W test180.c -o test180
test180.c: 関数 `main' 内:
test180.c:8: 警告: comparison of unsigned
                    expression<0 is always false
$gcc test180.c -o test180
$
```

符号なし数値をゼロより小さいか？と比較することは無駄です．これも警告されます(**リスト17**)．

コンパイルの結果を以下に示します．

```
$gcc -W test181.c -o test181
test181.c: 関数 `main' 内:
test181.c:10: 警告: comparisons like X<=Y<=Z
      do not have their mathematical meaning
$gcc test181.c -o test181
$
```

〔リスト16〕-Wオプションでエラーになる例(その3)(test180.c)

```
/*
 *-Wオプションでエラーになる例
 */
#include <stdio.h>
int main(void)
{
    unsigned    int i   =   9;
    if  (i  <   0   )   return  1;
    return  0;
}
```

〔リスト17〕-Wオプションでエラーになる例(その4)(test181.c)

```
/*
 *-Wオプションでエラーになる例
 */
#include <stdio.h>
int main(void)
{
    unsigned    int x;
    unsigned    int y;
    unsigned    int z;
    if  (x<=y<=z)   return  1;
    return  0;
}
```

〔リスト18〕-Wオプションでエラーになる例(その5)(test182.c)

```
/*
 *-Wオプションでエラーになる例
 */
#include <stdio.h>
const   int test(void);
int main(void)
{
    return  test();
}
const   int test(void)
{
    return  10;
}
```

これは文法的に意味のない比較なので，警告されます(**リスト18**)．

コンパイルの結果を次に示します．

```
$gcc -W test182.c -o test182
test182.c:5: 警告: type qualifiers ignored
                   on function return type
test182.c:11: 警告: type qualifiers ignored
                   on function return type
$gcc test182.c -o test182
```

関数の戻り値にconst指定しても，lvalueとして扱われるわけではないので意味がありません．警告されます．

-Wオプションで，以上のような細かい警告を出力することが可能になります．

● -Wdisabled-optimization

このオプションは，コンパイラが最適化することに非常に負荷がかかるようなソースを扱い，最適化を断念したときに警告を出すものです．たとえば一つの関数が数千行あったり，条件のネストが異常に深かったりした場合です．

〔リスト19〕浮動小数点値を==で比較する例(test183.c)

```
/*
 *浮動小数点値を=で比較する例
 */
#include <stdio.h>
int main(void)
{
    float   test1   =   1.0f/4.0f;
    float   test2   =   0.25f;
    if  (test1  ==  test2   )
    {
        printf("等価です¥n");
    }
    return  0;
}
```

〔リスト21〕構造体に対してパックを行っても，その効果がない場合の例(test185.c)

```
/*
 *構造体に対してパックを行っても,
 *その効果がない場合の例
 */
int main()
{
    struct str01
    {
        int x;
        char a, b, c, d;
    } __attribute__((packed));
    struct str02
    {
        char a;
        struct str01 data1;
    };
    struct  str02   data2;
    data2.a         =   'a';
    data2.data1.x   =   0;
    data2.data1.a   =   '0';
    data2.data1.b   =   '0';
    data2.data1.c   =   '0';
    data2.data1.d   -   '0';
    return  0;
}
```

〔リスト22〕生成されたアセンブラソース(test185.s)

```
    .file   "test185.c"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movb    $97, -24(%ebp)
    movl    $0, -23(%ebp)
    movb    $48, -19(%ebp)
    movb    $48, -18(%ebp)
    movb    $48, -17(%ebp)
    movb    $48, -16(%ebp)
    movl    $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (GNU) 3.3"
```

● -Wfloat-equal

OSの環境にも依存しますが，通常は浮動小数点値を“ == ”で比較することは危険です．このオプションはそのような計算式に警告を行います(**リスト19**)．

〔リスト20〕printfなどのフォーマットをチェックする例(test184.c)

```
/*
 *printfなどのフォーマットをチェックする
 */
int main()
{
    printf("%d¥n",9);
    printf("%a¥n",9);
    printf("",9);
    return  0;
}
```

コンパイルの結果を以下に示します．

```
$gcc -Wfloat-equal test183.c -o test183
test183.c: 関数 `main' 内:
test183.c:9: 警告: comparing floating point
                                with==or!=is unsafe
$
```

● -Wmissing-format-attribute

このオプションは-Wformatとともに使用し，printf系の関数に指定するフォーマット文字列をチェックするものです．フォーマット文字列が適切でない場合に警告します(**リスト20**)．

コンパイルの結果を以下に示します．

```
$gcc -Wmissing-format-attribute -Wformat
                              test184.c -o test184
test184.c: 関数 `main' 内:
test184.c:7: 警告: フォーマットはdoubleですが,
                    引数はdifferent typeです (引数2)
test184.c:8: 警告: zero-length printf
                                     format string
```

● -Wno-deprecated-declarations

このオプションは，関数属性の補足で解説します．

● -Wno-multichar

このオプションを指定すると，デフォルトで警告を出すマルチ文字定数を含んだソースに警告しなくなります．

マルチ文字定数を使うと可搬性に問題が起きるので，使わないほうがよいと思います．

● -Wpacked

構造体に対してパックを行っても，その効果がない場合に警告します(**リスト21**，**リスト22**)．

コンパイルの結果を以下に示します．

```
$gcc -Wpacked test185.c -S
test185.c: 関数 `main' 内:
test185.c:8: 警告: packed attribute is
                               unnecessary for`x'
test185.c:9: 警告: packed attribute is
                               unnecessary for`a'
test185.c:9: 警告: packed attribute is
                               unnecessary for`c'
$
```

〔リスト23〕構造体に対しパディングを行う例(test186.c)

```
/*
 *構造体に対しパディングを行う例
 */
int main()
{
    struct str01
    {
        char    a;
        int     b;
        char    c;
    };
    struct  str01   data1;
    data1.a         =   'a';
    data1.b         =   0;
    data1.c         =   'a';
    printf("%d¥n",sizeof(data1));
    return  0;
}
```

〔リスト24〕生成されたアセンブラソース(test186.s)

```
    .file   "test186.c"
    .section    .rodata
.LC0:
    .string "%d¥n"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movb    $97, -24(%ebp)
    movl    $0, -20(%ebp)
    movb    $97, -16(%ebp)
    movl    $12, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
    movl    $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (GNU) 3.3"
```

〔リスト25〕実行されないコードに警告する例(test187.c)

```
/*
 *実行されないコードに警告する例
 */
int main()
{
    struct str01
    {
        char    a;
        int     b;
        char    c;
    };
    struct  str01   data1;
    data1.a         =   'a';
    data1.b         =   0;
    data1.c         =   'a';
    printf("%d¥n",sizeof(data1));
    return  0;
    printf("実行されません¥n");
}
```

〔リスト26〕生成されたアセンブラソース(test187.s)

```
    .file   "test187.c"
    .section    .rodata
.LC0:
    .string "%d¥n"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movb    $97, -24(%ebp)
    movl    $0, -20(%ebp)
    movb    $97, -16(%ebp)
    movl    $12, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
    movl    $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (GNU) 3.3"
```

〔リスト27〕実行されないコードに警告する例(test188.c)

```
/*
 *使用していない値が
 *ある場合に警告する例
 */
int main()
{
    1 + 2 + 3;
    return  0;
}
```

〔リスト28〕生成されたアセンブラソース(test188.s)

```
    .file   "test188.c"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movl    $0, %eax
    leave
    ret
    .size   main, .-main
    .ident  "GCC: (GNU) 3.3"
```

● -Wpadded

GCCでは構造体の各要素を4バイト境界に配置します．配置できない場合にパディングしますが，その状態になったときに警告します(**リスト23**，**リスト24**)．

コンパイルと実行の結果を以下に示します．

```
$gcc -Wpadded test186.c -o test186
test186.c: 関数 `main' 内:
test186.c:9: 警告: padding struct to
                                  align`b'
test186.c:11: 警告: padding struct size to
                         alignment boundary
$ ./test186
12
```

```
$gcc -Wpadded test186.c -S
test186.c: 関数 `main' 内:
test186.c:9: 警告: padding struct to
                                  align`b'
test186.c:11: 警告: padding struct size to
                         alignment boundary
$
```

● -Wunreachable-code

ソース中に実行されないコードがあるときに警告します．このようなあからさまな例はともかく，デバッグ中に実行されないコードが出現することは珍しくないと思います．そのような場合に警告してくれます(**リスト25**，**リスト26**)．

コンパイルの結果を次に示します．

〔表4〕警告を要求・抑止するオプションのまとめ

バージョン2.95	バージョン3.3	
-fsyntax-only	-fsyntax-only	
-pedantic	-pedantic	
-pedantic-errors	-pedantic-errors	
-w	-w	
-W	-W	
-Waggregate-return	-Waggregate-return	
-Wall	-Wall	
-Wbad-function-cast	-Wbad-function-cast (C only)	
-Wcast-align	-Wcast-align	
-Wcast-qual	-Wcast-qual	
-Wchar-subscripts	-Wchar-subscripts	
-Wcomment	-Wcomment	
-Wconversion	-Wconversion	
	-Wdisabled-optimization	追加
	-Wendif-labels	追加
-Werror	-Werror	
-Werror-implicit-function -declaration	-Werror-implicit-function -declaration	
	-Wfloat-equal	追加
-Wformat	-Wformat	
-Wid-clash-len		廃止
	-Wformat=2	追加
	-Wformat-nonliteral	追加
	-Wformat-security	追加
-Wimplicit	-Wimplicit	
-Wimplicit-function -declaration	-Wimplicit-function -declaration	
-Wimplicit-int	-Wimplicit-int	
-Winline	-Winline	
-Wlarger-than-len	-Wlarger-than-len	
-Wlong-long	-Wlong-long	
-Wmain	-Wmain	
	-Wmissing-braces	追加
-Wmissing-declarations	-Wmissing-declarations	
	-Wmissing-format-attribute	追加
-Wmissing-noreturn	-Wmissing-noreturn	
-Wmissing-prototypes	-Wmissing-prototypes (C only)	
-Wmultichar		廃止
-Wnested-externs	-Wnested-externs (C only)	
	-Wno-deprecated -declarations	追加
	-Wno-div-by-zero	追加
	-Wno-format-extra-args	追加
	-Wno-format-y2k	追加
	-Wno-format-zero-length	追加
-Wno-import	-Wno-import	
	-Wno-multichar	追加
	-Wnonnull	追加
	-Wpacked	追加
	-Wpadded	追加
-Wparentheses	-Wparentheses	
-Wpointer-arith	-Wpointer-arith	
-Wredundant-decls	-Wredundant-decls	
-Wreturn-type	-Wreturn-type	
	-Wsequence-point	追加
-Wshadow	-Wshadow	
-Wsign-compare	-Wsign-compare	
-Wstrict-aliasing		追加
-Wstrict-prototypes	-Wstrict-prototypes (C only)	
-Wswitch	-Wswitch	
-Wtraditional		廃止
	-Wswitch-default	追加
	-Wswitch-enum	追加
	-Wsystem-headers	追加
	-Wtraditional (C only)	追加
-Wtrigraphs	-Wtrigraphs	
-Wundef	-Wundef	
-Wuninitialized	-Wuninitialized	
-Wunknown-pragmas	-Wunknown-pragmas	
	-Wunreachable-code	追加
-Wunused	-Wunused	
	-Wunused-function	追加
	-Wunused-label	追加
	-Wunused-parameter	追加
	-Wunused-value	追加
	-Wunused-variable	追加
-Wwrite-strings	-Wwrite-strings	

```
$gcc -Wunreachable-code test187.c -S
test187.c: 関数 `main' 内:
test187.c:18: 警告: will never be executed
$
```

● -Wunused-value

使用していない値がある場合に警告します．この例では計算値を変数に入れ忘れているのかもしれません（**リスト27**，**リスト28**，p.167）．

コンパイルの結果を以下に示します．

```
$gcc -Wunused-value test188.c -S
test188.c: 関数 `main' 内:
test188.c:6: 警告: statement with no effect
$
```

生成されたアセンブラソース上で，意味のない値は排除されています．

以上の警告を要求・抑止するオプションを**表4**にまとめます．

*　　　　*

次回は，「コード生成規約に対するオプション」の補足，「最適化オプション」の補足を行う予定です．

きし・てつお

小型・著作権保護機能を特徴とする

SDIOカード開発入門

第2回

SDIO規格の概要

山崎宣章

はじめに

市場では一般的に，SDカードといえば，SDメモリカードのことを指す場合が圧倒的に多い．そのメモリカード分野で，SDカードと競合しているコンパクトフラッシュでは，無線LANやBluetoothカードなどの製品をよく見かける．しかし現在のところ，残念ながらSDカードではこのような通信カードをほとんど見かけない．

これは，SDカードで無線LANやBluetoothカードが実現不可能だからというわけではない．SDメモリ規格では実現できないだけであって，SDIO規格では充分に実現可能なのである．

ここでは，このSDIOとはどのような規格なのか，SDメモリとSDIO両者の関係をふまえながら，SDIO規格の概要について解説する．

SDIO規格は，SDアソシエーション(http://www.sdcard.org/)により管理されている．規格の詳細に関しては，会員登録を行ったうえで公開される『SD Card Specification - Secure Digital Input/Output (SDIO)』を参照する必要がある．そのため，一般誌上では詳細まで触れることはできないが，その概要について説明を行う．

SDIOカードの基本的な概念

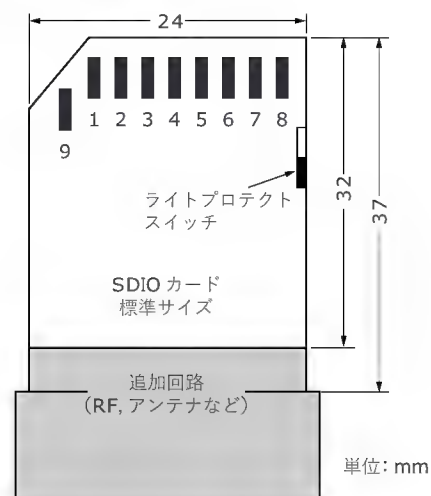
SDIOカードの基本的な概念の多くは，SDメモリより引き継がれている．中でもハードウェア特性に関しては完全な互換性を保持しており，形状，物理的なピン配置，電気的特性もまったく同じことから，SDカードサイズ以外に追加回路による出っぱりなどがなければ，外見や特性などでその違いを判断することは難しい(**図1**)．

機能面においては，SDメモリで実現しているホットプラグ(プラグ&プレイとも呼ばれているが，ここではホットプラグと呼ぶ)を採用していることで，SDメモリ同様，ユーザー側の使い勝手の良さをそのまま引き継いでいる．

また，スロットの規格に関しては，SDメモリとSDIOでは共通化されており，SDメモリのスロットにSDIOカードを挿入することが可能である．また，電気特性やピン配置といった基本的な特性も共通化されていることから，電気的に破壊されることもない．しかしながら，SDメモリ用スロットでは，SDIOを動作させることはできない．その理由は後述するが，ニュアンス的にはSDIOが上位互換のイメージとなるためである(まだ製品化はされていないが，SDIOではSDメモリの機能をもたせることが可能)．

SDメモリとSDIOカードの大きな違いは，IO(Input/Output)通信機能の有無にある．たとえば，802.11b無線やBluetoothなどといった通信デバイスや，小型カメラのような映像デバイスなどをSDカードで実現できるのが，SDIO規格なのである．

その通信機能実現のため，SDIO規格ではSDバス上を行き来するSDコマンドを通信向けに拡張し，SDメモリ規格では実現し得なかった複雑な通信処理を行えるようにした．反面，このコマンド拡張により，これまでのSDメモリ用コマンドとは互換性が保てなくなった．そのため，SDメモリしかサポートしていないSDホスト機器には，SDIOカードを認識させることができなくなっている．SDホスト機器のファームウェアをアップグレードするなど，ソフトウェア的な対応でSDメモリ用の機器をSDIOに対応させることは困難になっている．SDIOカード

〔図1〕SDIOカードの外形図

を使用する場合には，初めからSDIOに対応しているSDIOホストデバイスをあらかじめ準備しておくことが不可欠である．

SDバス

SDバスとは，機器側のSDスロットとSDカードの間で信号を橋渡しをするバスである．全部で9本の信号線で構成されており，クロック，電源電圧，グラウンド，コマンド，データ，割り込み要求がある（**表1**）．動作周波数（SDクロック）は0～25MHzで，データ転送レートは，最高100Mbpsである．SDバス自体は，SDメモリとの互換性も保たれており，SDメモリのバス同様，3種類のバスモードをもっている．それぞれのバスモードによりピン配置，最高データ転送レートも変化する．

▶SD 1ビットモード

もっとも標準的なバスモードであり，すべてのSDIOカードは必ずこのバスモードに対応していなければならない．CMD（コマンドライン）ピン1本で，SDホスト側からのコマンド（命令）とSDカード側からのレスポンス（応答）をやりとりし，DATA（データライン）ピンでデータのリード/ライトを行う．このデータ用のピンの割り当てが1本であることから，「SD 1ビットモード」と呼ばれている．

▶SD 4ビットモード

SD 1ビットモードのデータピンを拡張したバスモードであり，その名のとおり，データのリード/ライト用に4本のピンを使うモードである．この4本のデータピンを活用することで，100Mbpsの高速転送を実現可能としている．

▶SPIモード

IC間通信に広く採用されているシリアル通信仕様の一つであるSPIをサポートしているモードで，コマンド/アウトプット用に1ピン（DO），レスポンス/インプットに1ピン（DI）を使用している．

昨今，“高速アクセス”との謳い文句で販売されているSDメモリは，SD 4ビットモードでのアクセスで実現されている場合が多い．

SDコマンドの概念

SDコマンドの基本構成としては，SDホスト側からのコマンド（またはデータのライト）と，SDカード側からのレスポンス（またはデータのリード）で成り立っている．

SDコマンドの種類は，CMD0～CMD59まで用意されており，SDメモリとSDIO，またSDモード（SD 1ビットモード/SD 4ビットモード）とSPIモードとで，使用できるコマンドが異なってくる．

- SDモード時：SDメモリ用コマンド32種類/SDIO用コマンド7種類
- SPIモード時：SDメモリ用コマンド29種類/SDIO用コマンド5種類

ここでは，本文でとくに明記しない場合は，SDモード時のSDIO用コマンドをSDIOコマンドと呼ぶことにする．

SDIOコマンドの種類の数は，SDメモリと比べコマンド数が減っている．SDメモリと比べて使用される機能の種類は増えているものの，コマンドの種類が減少しているのである．これは，SDIO用で追加となったCMD52，CMD53でSDメモリのほとんどのコマンドが実現できることになったためである．このコマンドの具体的な機能に関しては後述する．

SDIOのFunction

SDIOにはさまざまなデバイスが接続できる．メモリはもちろん，無線LAN，UART，PCMCIA，Bluetooth，GPS，カメラなどのいろいろなデバイスを接続できる．これは，SDIO特有の“Function”という概念により実現されている．

単純にFunctionとは，SDIOでのI/Oポートそのものを指しており，1枚のSDIOカードでFuction0～7までの最高8個のFunction（I/Oポート）をもつことが可能である．しかしながら，Function0はすでにSDメモリに予約されているため，SDカードの設計者は，Function1～7を使用することになる（**図2**）．

接続されたデバイスを使用するときは，このFunctionに割り当てられたデータアドレスに対し，動作電圧，動作周波数，接

〔表1〕SDバスモードと信号線

ピン	SD1ビットモード		SD4ビットモード		SPIモード	
1	N/C	未使用	CD/DAT[3]	カード検出/データライン[3]	CS	カードセレクト
2	CMD	コマンドライン	CMD	コマンドライン	DI	データインプット
3	V_{SS1}	グラウンド	V_{SS1}	グラウンド	V_{SS1}	グラウンド
4	V_{DD}	供給電圧	V_{DD}	供給電圧	V_{DD}	供給電圧
5	CLK	クロック	CLK	クロック	SCLK	クロック
6	V_{SS2}	グラウンド	V_{SS2}	グラウンド	V_{SS2}	グラウンド
7	DATA	データライン	DAT[0]	データライン[0]	DO	データアウトプット
8	IRQ	割り込み要求	DAT[1]	データライン[1]	IRQ	割り込み要求
9	RW	リードウェイト	DAT[2]	データライン[2]	NC	未使用

続されるデバイス先のI/Oアドレスなどを設定しておき，このFunctionのアドレスを経由して，接続されたデバイスを制御しなければならない．たとえばFunction2に割り当てられているPCMCIA経由で，PCMCIA用のPHSモジュールを制御する．

SDホスト機器に搭載されているデバイスドライバの動作としては，SDIOコマンドを用いて，Function内のアドレスを経由して，接続されたデバイスを制御するためのI/Oへアクセスすることになる．

SDIOの初期化

SDカードと呼ばれるカードには，正確には4種類のカードが存在している．SDメモリカードとMMCカード，SDIOカード，SDIO/メモリコンボカードの4種類である．またSDバスの規格としては，SD 1ビットと4ビット，そしてSPIモードの3種類が存在している．そのうえ，Function(SDIOのI/Oポート)にはさまざまなデバイスが接続されている．

SDカードは，すべてのSDスロットに挿入することができ，すべてのカードがホットプラグに対応している．

これは，ユーザーがSDスロットにSDカードを挿入した瞬間に，SDホスト機器がSDカードの種類を認識する必要があり，なおかつSDIOカード(コンボカードも含む)であった場合，Functionにどのような機器が接続しているのかを，自動的かつ即座に認識しなければいけないことを意味している．

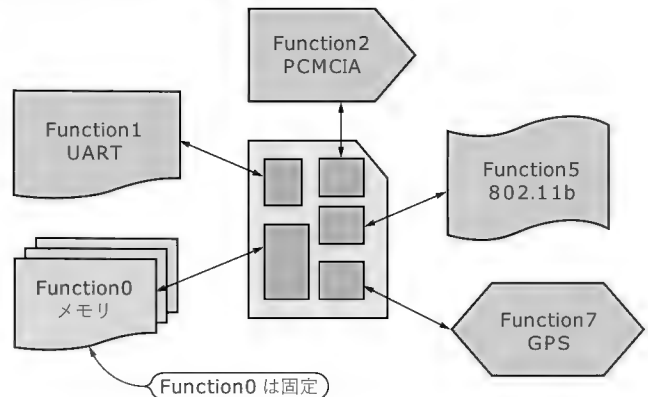
これらを実現するために，SDIO規格では初期化に関し，厳密で細かいフローチャートを用意している．実際に，SDカードの初期化は次のプロセスで行われる．

(1) バスモードの決定

SPIモードとSDモード(1ビット/4ビット)とでは，ピン配置も異なり，コマンド体系も異なっているため，SDカードの初期化では，まずこのバスモードの決定を行い，SDカードとの通信ができる環境を最優先で構築する．具体的には次のようにバスモードを決定する．

まず，SDホスト機器にカードが挿入されると，カードに対しバスパワー[注1]の供給を開始する．

そしてカードに対し，CMD0(コマンドゼロ)を発行する．このコマンドは，単にカードからの返答(エコー)が戻ってくるかの確認用コマンドであり，カード側から返答があり，なおかつSDバスの1ピン(**表1**参照)が，“L”レベルであったときはSPIモードで動作させる．“L”レベルでなければSDモードとして動作させる．

SDモードの1ビット/4ビットの選択は，コマンド体系が共通であるため，ここではとくに設定する必要はない．デフォルトの1ビットのままで設定が継続され，4ビットが必要であれば，後で任意に設定を行う．CMD0からの返答がなかった場合は，再度カードの挿入を検出するまではカードへのアクセスは行われない．

注1：外部電源を必要とせず，SDバスから供給される電源．

〔図2〕Function接続例

(2) SDカードの種類の決定(SDモード使用時)

物理的な初期化の準備が終了した時点で，SDホストとSDカードの間ではコマンドが使用できる環境が整えられている．そこで，次はカードの種類(SDメモリ，MMC，SDIO，コンボ)の判定に移る．

SDホストでは，挿入されたカードの把握を行うために，SDコマンドを使用して，そのレスポンス(もしくはタイムアウトエラー)の内容を，カードの種類の判断材料としていく(詳細は，後述のSDコマンドの項で解説)．

実際，この初期化のプロセスには，最低7種類のコマンドと4種類の変数と5種類のフラグを使用して行われる(初期化の手順は，SDホスト機器に搭載されているSDホストチップにより，手順が異なる場合がある)．

(3) SDIOカードの個体情報の収集

SDカードの種類が確認できたことで，SDホストは次に挿入されたカードの個体情報の収集を実施する．この個体情報とは，挿入されたカードにどのようなデバイスが搭載されているか，必要な電流やSDバスのクロック周波数など，カード固有情報なども含まれており，この初期化が正常に終了した時点で，SDホスト機器はカードの情報をすべて掌握することになり，初期化を終了する．

この初期化のプロセスは，一般的にカードが挿入されて1秒にも満たない間に終了する．ホットプラグと，この高速な初期化により，ユーザーは使いたいときにカードを差し，使わなくなったら抜くという動作を安定して行えるようになっている．

SDIOのアドレス概念

SDIOのアドレス空間は大きく分けて，2種類に分かれている(**図3**)．

一つは，CIA(Common I/O Area)と呼ばれており，SDIO

〔図3〕SDIOのアドレスマップ

カードの基本となる情報が格納されている．このCIAに格納されている情報で，SDIOカードの機能がすべて決定される．また，このCIA内部でも複数の空間に分かれており，その空間にカードで使用される各Functionごとの細かいパラメータなどを記載する．

もう一つは，CSA(Code Storage Area)と呼ばれており，SDホスト側がそのSDカードを使うために，ドライバやアプリケーションを必要としている場合の拡張空間であり，このCSA空間にそれらのデータを格納しておき，ホットプラグ直後にカードからSDホストへ，そのデータを転送させることができる．

● CCCR(Card Common Control Register)

SDカードの全機能を司るCIA空間の最上位に位置しているのが，このCCCRである．このレジスタはFunctiono(カード自体)に対する基本情報を格納する．本来は，SDメモリカード時代の基本情報を格納していたレジスタであったが，SDIOとなり複数のFunctionをサポートできるようになったことから，SDIOではカード全体の情報を格納するレジスタとなっている．

具体的には，そのカードが対応しているSDIO規格のバージョンや，そのカードで使用可能なFunction数およびFunction名(1〜7)，後述のCIS空間のポインタ(参照先アドレス)などの情報を格納する．また，各Functionの稼働状況も常に更新された状態で格納している．

● FBR(Function Basic Register)

CCCRがFunctiono専用レジスタなのに対し，FBRは，Function1〜7のSDIOに接続される各種デバイス用の基本情報を格納するためのレジスタである．もしカード設計者が，Functionを一つしか使用しないのであれば，FBRは1種類だけを設定しておけばよいが，複数Functionを使用する設計をしているのであれば，使用されるFunction分だけのFBRを設定する必要が生じる．

具体的に格納すべき情報とは，接続されているデバイスの種類(メモリ，UART，GPS，PHSなど)，データ転送の際のデータブロックのブロックサイズや，前述のCCCR同様，CIS空間へのポインタ(参照先アドレス)などを格納する．

また，Functionに接続されているデバイスを使用するにあたり，SDホスト機器側にドライバやアプリケーションが必要で有れば，それらを格納しているCSA空間へのWindowと呼ばれるポインタも格納することができる．

● CIS(Card Information Structure)

SDホストやSDカードに必要とされる具体的な情報や，Functionに接続されたデバイスとの橋渡しを行う．カードとしての実作業が行われる空間であり，前述のCCCRやFBRから参照されてくる．基本的には，カード設計者が独自に使用できる空間ではあるが，CCCRからの参照されるCISのFunctionoの空間に関しては，例外となり，SDIO規格に則った記述が必要になる．

CCCRから参照されてくる空間は，SDホスト機器に必要最低限の情報を格納するレジスタであり，挿入されたカードに必要な電流やSDバスのクロック周波数，カード固有情報など，そのカードを使用するために必要な仕様を格納をする．また，記述フォーマットはSD規格で厳密に定められており，必ず記載をしておく必要がある．

前述のSDカードの初期化の際は，SDホスト機器が，このCCCRを参照し，当空間のデータを参照するため，記述に誤りがあると初期化自体が正常に終了できず，SDホスト機器がカードを正常に認識することができなくなる．

FBRから参照されてくる空間は，基本的にはカード設計者が自由に使用(レイアウト)することができ，アドレスそれぞれに機能をもたせることが可能である．たとえば，Functionに通信用デバイスが接続されているのであれば，送信用アドレス，受信用アドレスをCIS空間内に設定する．この送信/受信用アドレスは，接続されている通信デバイスの送信/受信アドレス(またはポート)にリンクさせておく．SDホスト機器からはSDコマンドを使って，CIS内のこの送信/受信用アドレスにアクセスを行うことで，Functionの先に接続されている通信用デバイスでのデータの送信/受信を行うことができるのである．

SDIOコマンドの概要

SDIOコマンドの原則として，SDIOホスト機器側からコマンドを発行して，SDカードよりレスポンスを返す．またデータ

に関しては，すべてホスト側からの書き込み/読み込みで行われ，カード側からホストに対しての書き込み動作は行うことができない．唯一のカード側からホストへの連絡方法としては，割り込み要求の使用のみとなる（割り込み要求後に，ホストからデータの要求をしない限りは，カードはいつまでもデータを保持したままとなる）．

すべてのSDカードでは，共通のSDコマンドフォーマットが決まっている．実際SDバス上でやりとりされるものは，コマンド，レスポンス，データの3種類だけとなる．SDIOコマンド体系は冒頭でも記述したが，SDカード（メモリ/SDIO）とSDモードによってそれぞれ異なるため，ここでは，一般的なSDIOカードとSDモード（1ビット/4ビット）の組み合わせでのコマンド体系を中心に紹介する．

● SDコマンドでの初期化

SDコマンドでの初期化に関しては，前述のSDIOの初期化に重複するが，SDコマンドでの初期化作業に関しては，非常に興味深いシーケンスを実施する．SDコマンドには，各種SDカードごとに特有のコマンドをもっており，その特有のコマンドをカードに対し次々と発行する．カードは対応していないコマンドを受け取ると，エラーのレスポンスを返すか，レスポンスそのものを返さないため，SDホスト機器はその反応を見て，挿入されたSDカードの種類を判断する（詳細のコマンド名に関しては，SDアソシエーション刊行のSD Card Specificationを参照）．

SDカードの判断が終了したら，SDホスト機器はカードに対し，通常のデータリードコマンド〔CMD52（コマンド52）〕を使用して，CCCR（カード情報），CIS（詳細情報）へアクセスを行い，必要な消費電力やSDバスの周波数の情報を入手し，初期化を終了させる．これ以降で，SDホスト機器はSDカードを本格的に使用することができる．

● SDメモリコマンドからSDIOコマンドへ

実際のSDIOで使用されるコマンドは，2種類だけである．もともとはSDメモリの規格がベースとなっているため，SDメモリで個別のコマンドとして存在していたものを，すべて直接アドレスにアクセスさせることで，二つのコマンドに集約してしまったのだ．この二つのコマンドは，単純にアドレスに対しての読み込み/書き込みを行うだけのコマンドであり，2種類というのは，バイト単位での読み書きか，ブロック単位での読み書きかの違いになる．

▶ **CMD52（コマンド52）**

CMD52とは，一つのアドレスに対し，一つのデータを読み書きするためのコマンドである．このコマンドのメリットとしては，SDバス上のCMDピンのみでアドレスの読み書き行える点である．おもに初期化や1行だけのコマンドを発行するような場合に適している．

注2：シイガイズ（株） http://www.c-guys.jp/

▶ **CMD53（コマンド53）**

CMD53はブロック転送を可能にしたコマンドで，一つのコマンドで複数バイトのデータを転送できる．この際，データのブロック内のバイト数に関しては，FBR（カードの詳細情報）レジスタで設定する．コマンド自体はSDバス上のCMDピンを使用し，データ伝送にはSDATピンを使用する．

実際のアプリケーション（SDホスト機器）の動作としては，カードが挿入されたときにハード的な初期化を行い，SDバスが開通した時点でカード情報を入手し，これまでに記述したCIA内に配置されたCCCR，FBR，CIS（場合によってはCSA）の各レジスタに対し，CMD52/53を用いてアクセスを行う形となる．

アプリケーション開発においては，初期化とCIAの各レジスタの役割さえ把握できれば，2種類しかないコマンドでの開発となるため，複雑なコマンド体系を覚える必要性はない．

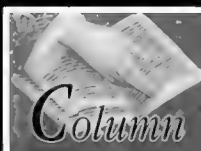
まとめ

以上のように，SDバス，SDコマンドについて解説を行ってきた．より詳細な情報に関しては，SDアソシエーション（http://www.sdcard.org/）に会員登録を行ったうえで，公開されている『SD Card Specification - Secure Digital Input / Output（SDIO）』を参照いただきたい．会員登録を行うことで，SDIOだけの情報ではなく，SD-PHS，SD-Bluetooth，SD-802.11bなどの参考資料も入手することができる．

実際にカード設計を行う際，いちばん重要で問題が発生しやすいポイントは，初期化のシーケンスである．現在，市場にあるSDホスト機器に搭載されているSDホストコントローラは，製品ごとに多少の方言をもっているようで，すべてのSDホストコントローラで安定して初期化を終了させるためには，プログラム的にいろいろ気を使う必要がある．そんな中，シイガイズ社[注2]が製造しているCG100 SDIOコントローラのように，ユーザーが初期化に関していっさい気にすることなく，すでにUARTやPCMCIA機能がFunctionに割り当てられていて，無線LANやカメラなどのデバイスに集中してカード開発を行えるような製品も販売されている．

今後は，このような製品を応用したSDIOカード製品が市場に多く出回ってくることだろう．SDカードのうたい文句である「小さい」，「省電力」，「ホットプラグ」は，携帯機器における重要な要素でもあるからである．

そうなってくれば，ちょっとした外出でも，PDAと各種SDIOカードを入れたPCMCIA用のケースを一つもって出れば，「歩くリトルオフィス」が実現できる．大昔からいわれ続けている万能携帯機器も，実現できそうでできていないと思っているが，SDIOが本当に実現してくれるのではないかと思っている．

やまざき・のぶあき 松下テクノトレーディング（株）
マーケティンググループ マーケティングチーム

IEEE802.11b対応 SDIO無線LANカード

平山勝啓

実際の市販SDIOカードの例として，IEEE802.11b対応のSDIO無線LANカード「SD-Link11b」(シイガイズ)を紹介します．

● SDIO無線LANカード　SD-Link11b

SD-Link11b(**写真A**)は小型のSDIO無線LANカードでノートPC，PDA，デジタルカメラ，携帯AVプレーヤ，携帯電話などあらゆるモバイル機器において，さまざまなライフステージで無線LANへの接続を可能にします．また，強力なパワーセーブモードにより低消費電力で待ち受けを行うことができます．SD-Link11bはSDIOコントローラデバイス「SD-Pathファミリ CG100」(シイガイズ社製)を搭載しています．PocketPC2002/2003，Windows CE，Windows XP，Palmなどの汎用OSへの適応はもとより，最近SDIO対応が加速的に促進されているモバイルコンシューマ商品をターゲットに，さまざまな組み込み向けOSへの対応を図っています．**表A**にSD-Link11bの基本仕様を，**図A**にブロックダイヤグラムを示します．

▶パワーマネージメント

一般的なPCの無線LAN接続時の稼働状況分析によると，約25%が実際にデータ転送で稼動し，残り75%はアイドル(待ち受け)になっています．モバイル機器の場合，この比率はさらに広がり，電話などへの応用ではアイドル時間が支配的になります．このようにモバイル機器の特質に合わせたパワーマネージメントを提供し，システムとしての消費電力を最低限に抑えることが，モバイル市場では期待されています．SD-Link11bでは2種類のパワーセーブモードを提供し，アプリケーションに応じて対応させることが可能です(**図B**).

1) パワーセーブモード

パワーセーブモードでは，IEEE802.11b規格に定義される一定の受信データパケット信号(約100ms間隔)に同期させてRF回路を起動させ処理を行う場合，間欠動作を利用することで低電力化を実現できます．この間欠動作の時間も，アプリケーションに合わせて制御することが可能で，さまざまな用途に応じた消費電力の削減が可能となります．

2) ディープスリープモード

ディープスリープモードでは，必要最低限の回路機能のみを残し，他はシステムクロックさえも停止させるモードです．これにより消費電力は1mA以下(暫定)を実現し，このカードを機器に挿入したままでも電力消費を最少にできるしくみになっています．ディープスリープモードからの復帰はホスト側からの制御で行われ，およそ1秒以内で通常動作に復帰します．

● SDIOカードのラインナップの今後

筆者の会社(シイガイズ)では，第一弾としてSDIO無線LAN SD-Link11bを発表しましたが，今後も引き続き，SD-Bluetooth，SD-FMRadio，SDカメラなどの新カード製品を順次リリースする予定です．

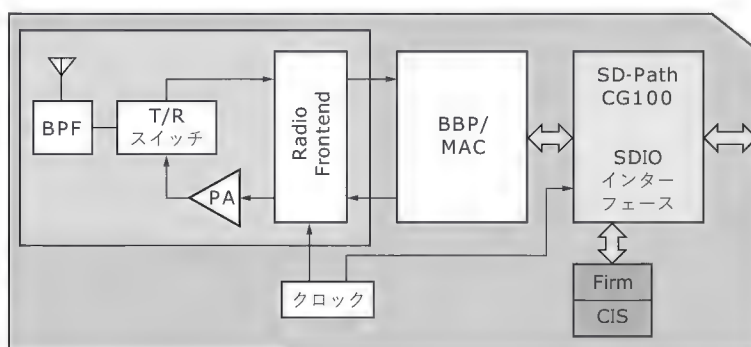
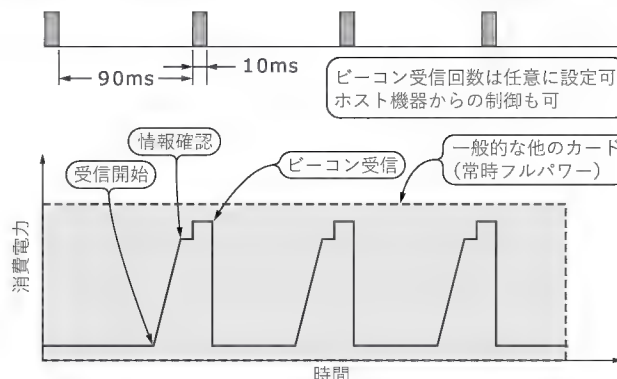
なお，2003年10月に開催されるCEATEC Japanの電子部品デバイスゾーン(ブース番号6D75)に出展し，SDIOソリューション技術および各種SDIOカード製品を展示する予定です．

ひらやま・かつひろ　シイガイズ(株)　http://www.c-guys.jp/

〔写真A〕SD-Link11b外観

〔表A〕SD-Link11bの基本仕様

インターフェース	SDIO標準規格Ver1.0
無線LAN仕様	802.11b
接続方式	インフラストラクチャモード，アドホックモード
伝送距離	屋外約400m 屋内約100m(環境条件による)
消費電力	動作時(平均260mA)，待ち受け時(3mA以下)，ディープスリープモード時(1mA以下)
セキュリティ設定	WEP(64ビット/128ビット)
外形寸法	55mm(長さ)×24mm(幅)×2.1mm(厚さ)

〔図A〕SD-Link11bカードのブロックダイヤグラム

〔図B〕SD-Link11bにおけるパワーセーブモードの概念図

組み込みLinuxをとりまく世界

第3回 「組込みLinux評価キット」(ELRK)を使ったWebサーバの構築

渡辺武夫

前回(2003年9月号),「組込みLinux評価キット(Embedded Linux Reference Kit : ELRK)の概要を解説した．今回は同キットを使って，実際の開発を行ってみる．

構築する内容と完成イメージ

ターゲットハードウェアとして，ARM7系CPUを搭載した組み込み向け小型CPUボード「Armadillo」(アットマークテクノ社)を用いて開発を行う．また，構築内容としては，同ボード単体で動作するWebサーバを構築し，ホームページの公開ができるものを作ってみる．

ELRKでは，ターゲットボードのシリアルとEthernetが標準で動作する．したがって，ネットワークについてはとくに気にする必要はない．それと別に考えなければならないことは，サーバには各種データファイルを保存する領域があり，それに対してデータ更新ができるという機能がなければならないということである．Armadilloには CF カードスロットがあり，コンパクトフラッシュ(CF)を装着することでストレージを設けることができるので，それをLinuxの各種ファイル保存場所兼HTMLファイル収納場所とする．また，ホームページの更新や各種ファイル操作を行うために，FTPによるファイル送受信，TELNETによるリモートログインができる環境とする(**図1**).

開発環境の構築

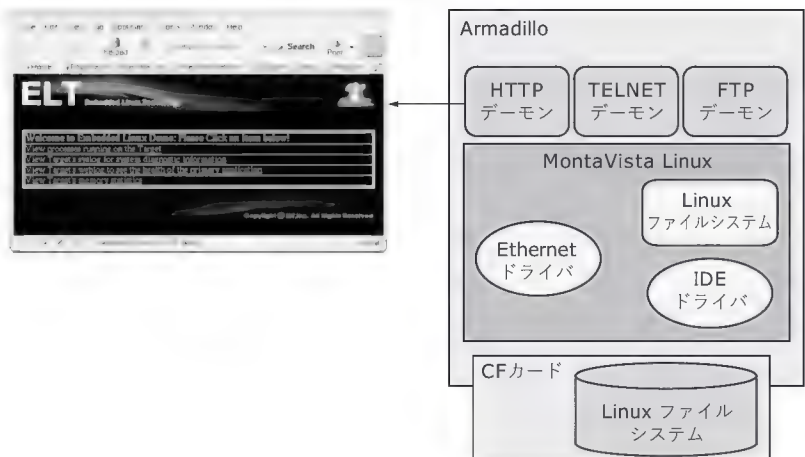
最初に，開発環境をホストコンピュータにインストールしなければならない．インストール方法については，製品同梱のマニュアルに記述されているので省略する．また，以降にさまざまな同梱ツールを使用するが，その起動方法などもマニュアルに記載されているので，そちらを参照いただきたい．インストールが完了した後のディレクトリ構成について重要となる点のみ，おさらいをかねて，**図2**に記しておく．以降，このディレクトリ構造を基準に話を進めていく．

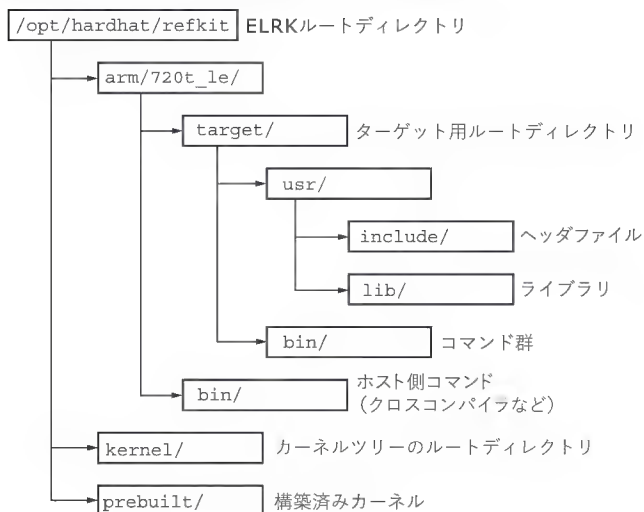
カーネルの再構築

まず，サーバとして動くのであれば，静的なIPアドレスをもつのが一般的である．それに対しELRKでは，ターゲットボードのIPアドレスはDHCP/BOOTPサーバを参照して設定するようになっているので，とりあえずこれをやめ，固定IPアドレスにすることにする．方法はいくつか考えられるが，Linuxでもっとも依存性が少ない方式で行ってみる．

- カーネルコンフィグレーションでの静的なパラメータ埋め込み

Linuxでは「カーネルコマンドストリング」という名称で，動作設定をアスキー文字列指定で行える．その設定内容にIPアドレス設定があるので，今回はそれを活用する．具体的な操作は，xconfigを用いたGUIで行える．まずは，基準とするconfigファイルを決め，カーネルソースのルートディレクトリに`.config`というファイル名でコピーする．今回は，armadillo-rt(リアルタイムオプション付き)を使った．次に，同ディレクトリでxconfigを起動させ，コンフィグレーション画面を立ち上げる(**図3**).

〔図1〕Armadilloを使ったWebサーバの完成イメージ

〔図2〕インストール完了後のディレクトリ構成

〔図3〕コンフィグレーション画面

この中の“Default Kernel command string”がこれにあたり，設定を行いたい内容を記すのみでLinuxの動作を設定できる．この部分に今回の目的となるIPアドレス指定文字列を入れてみる．また，同時に，ルートファイルシステム先(NFSマウント先)も指定しておく．

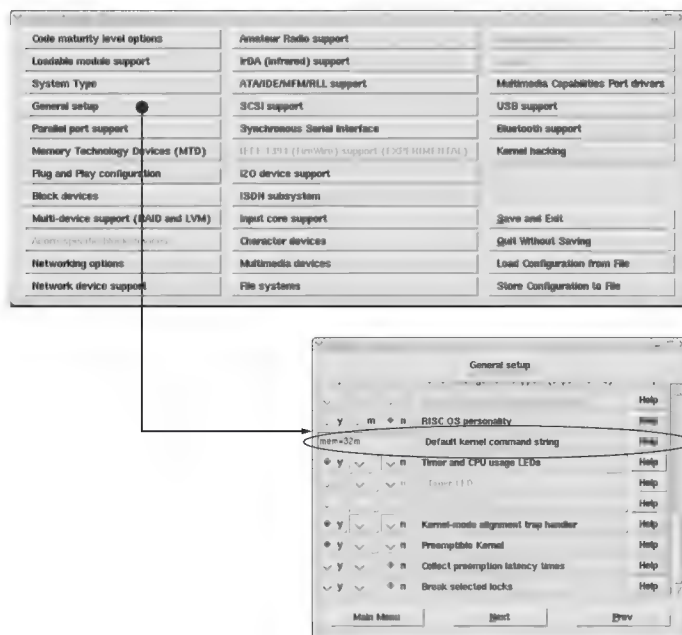
```
mem=32m ip=XXX.XXX.XXX.XXX nfsroot
        =YYY.YYY.YYY.YYY:/opt/refkitarm/
                            720t_le/target
```

XXX.XXX.XXX.XXX：ターゲットボードのIPアドレス
YYY.YYY.YYY.YYY：NFSホストのIPアドレス

最後に，この設定を保存してxconfigを終了させ，Linuxカーネルイメージをビルドさせると，起動設定を変更できる注1．

実際に再構築されたカーネルをArmadilloにダウンロード/実行してみて，IPアドレスが固定になったかどうかを確認してみる．正しく構築されていれば，起動時のメッセージからBOOTP要求が外れ，指定したIPでethoデバイスが起動したことを示す内容と，ルートファイルシステムのマウント先が同様に指定されたIPアドレスを参照していることが確認できる．

▶補足

同梱マニュアルでは，カーネルをダウンロードし，FISに保存する操作を連動して記している．もちろん，この方法であれば，ターゲットボードの起動ごとにカーネルイメージをダウンロードする必要はないが，今回のように，再構築されたカーネルの動作確認というのであれば，FIS保存を省略することもできる．つまり，カーネルをシリアルでダウンロードした後，EXECコマンドで起動してしまっても，問題はない(ただしボード起動ごとにダウンロードが必要だが)．

アプリケーションの構築

さて，カーネルの設定と構築はいったん中断し，今度はアプリケーションの構築を行う．今回の目的から，構築しなければならないアプリケーションは，Webサーバ/TELNETサーバ/FTPサーバの3種類となる．まず，ELRKのArmadillo用には，Webサーバ機能と，TELNETサーバ機能がすでにあるので，今回はFTPサーバの構築にのみ的を絞って説明する．

● ソースコードの入手

今回は，Debianからソースコードを入手してみた(`http://www.debian.org/`)．Debianのサイトからftpdで検索をかけて“wu-ftpd”を探し出し，これを使う．

● コンパイル環境の設定

基本的に，セルフ環境(コンパイルしたコンピュータ上で動作)が標準なので，クロスコンパイルができる環境を構築しなければならない．これについて正直なところ，すべてに共通な方法は，現時点ではない．したがって，入手したソースコードの構築環境を逐次確認しながら，クロスコンパイルができるような構成としなければならない．とりあえず今回は，次のような方法でコンパイルを行った．

注1：RedBootで指定するブートパラメータとの競合について
今回のArmadilloでは，ブートローダであるRedBootのLinux起動機構(EXECコマンド)で，起動パラメータを設定できる．ただし，これと上記で記したパラメータ設定を同時に使うことはできない．したがって，カーネルコンフィグレーションでカーネルコマンドストリングを設定した場合，EXECコマンドでの起動パラメータ設定(オプション-c)は使わないこと．

〔図4〕PSコマンドによる確認

```
# /root/ftpd -S                    ← ftpdの起動
# ps -aux
  PID  Uid     Stat Command
    1 root     S    init
    2 root     S    [keventd]
    3 root     S    [ksoftirqd_CPU0]
    4 root     S    [kswapd]
    5 root     S    [bdflush]
    6 root     S    [kupdated]
    7 root     S    [mtdblockd]
    8 root     S    [rpciod]
   33 root     S    syslogd
   37 root     S    klogd
   41 root     S    /usr/sbin/inetd
   45 www      S    /usr/sbin/thttpd -C /etc/thttpd/thttpd.conf -i /var/run/tht
   51 root     S    ftpd: accepting connections on port 21  ← ftpdが起動し，ポート番号21で待機中であることを明示
   55 root     S    -sh
   61 root     R    ps -aux
#
```

STEP1：ホストのx86/Linux上で動作させるものとして環境設定（セルフ環境としてconfigureを実行）

STEP2：Makefileの各所において，次の項目を変更

クロス開発環境コマンドパス：

`/opt/hardhat/refkit/arm/720t_le/bin`

コンパイラ名称：`arm720t_le-gcc`

ライブラリ保管場所：

`/opt/hardhat/refkit/arm/720t_le/target/lib`

ヘッダファイル保管場所：

`/opt/hardhat/refkit/arm/720t_le/target/usr/include`

STEP3：依存機能（今回はFTW機能）がない環境に変更（`HAVE_FTW`を外す）

● コンパイル

コンパイル環境の設定が終了したら，あとはMakefileを実行し，実行ファイルを生成する．結果として，いくつかのバイナリが完成するが，今回は最小限の動作設定にとどめ，ftpdのみを使うことにする．

● 動作させる

できあがったバイナリを動作させるために，ターゲットボードのファイルシステムにftpdをコピーしなければならないが，ELRKの環境では，ターゲットのファイルシステムは，ホストコンピュータの所定ディレクトリを使っているので，単純にホストコンピュータ上でファイルコピーを行うだけで，結果としてターゲットボードで読み込む状況とすることができる．つまり，できあがったftpdをホストコンピュータ上にある所定ディレクトリ（`/opt/hardhat/refkit/arm/720t_le/target/root`）の下にコピーするだけということである．

コピーが完了したら，ターゲットボードのLinuxにログインし，次のようにタイプすることでFTPサーバ（デーモン）を起動させることができる．

`/root/ftpd -S`

実際にプログラムが動作しているかどうかは，PSコマンド（`ps -aux`）で確認できる．また，ホストコンピュータからFTP接続を試みてみるのも，動作試験となるだろう．**図4**に，実際に起動させた状況でのログを示す．ftpdは起動が成功すると，

Column1

オープンソースのクロス開発

今回はオープンソースであるwu-ftpdをインターネットから入手し，クロス開発に適用してみたが，現状の多くのオープンソースはクロス開発に適したものとはなっていない状況がある．これは，提供者たちが純粋に，自分で保有しているLinuxコンピュータで動作するものとして開発しているため，結果としてセルフ開発が基準となっているのが最大の理由と思う．またLinuxの歴史において，クロス開発自体はそれほど重要視されていないのも事実で，極論でいうと，ターゲットボードでもLinuxが動作しているのだから，その上でコンパイルやデバッグも行うのが簡単ではないか，ということになる．

したがって，オープンソースで入手した各種ソフトウェアを，自分が考えているターゲットボード上で動作させるためにクロス開発するというのは，じつはたいへんなことだと思う．実際に試したが，一般手法（Configureツールが自動で環境設定してくれる）が通用しない（もしくは，見た目は成功しても実際にコンパイルしてみたらエラーだらけ）のがあたりまえといっても過言でないのが現実で，結果としてはあれこれ試行して，手探りで進んでいき，やっとクロス構築ができた．まあそれでも，自分で初めからすべてを作るよりは時間短縮できたのだから，そう考えると，オープンソースの価値があるといえるのだろう．

クライアントからのネットワークのポート番号21で接続待ち状態となるはずだが，psコマンドでそれが実現されていることがわかるはずである．

● 自動起動の設定

このままだと，Linuxを起動するたびに手作業でftpdを起動しなければならない．したがって，作成したftpdをLinuxの起動にともなって自動で起動しておく設定をする必要がある．これ自体はとくに難しいことではなく，今回の環境の場合，ターゲット用ルートファイルシステム内の起動スクリプトファイル“/etc/rc.d/rcS”の最後に次の行を入れるだけで実現できる．

```
Echo "Start FTPD"; /root/ftpd -S;
                                  echo "done."
```

● 補足

ftpdの起動についてはこれで終了だが，実際にFTP接続を行うためには，ログインアカウントにパスワードを設定しておく必要がある．方法としては，ターゲットボードのLinuxにログインした状態で，passwdコマンドを実行することにより，パスワード設定が可能となる．

デバッグ

今回は非常に小さいプログラムで，また比較的問題のないものを実装したので，「デバッグ」という言葉は出てこなかったが，実際の作業では，プログラムが正しく動作しないため，デバッグ作業を行うことが日常茶飯事だと思う．ELRK環境でデバッグを行うには，次の2種類の方法が考えられる．

● 手法1：printfを用いたデバッグメッセージ出力方法

これは従来からあるもっとも原始的な手法の一つである．単純にプログラム中にprintf()を埋め込み，メッセージ出力によりプログラム通過点の確認をする，もしくは変数などの内容確認を行う方法である．

● 手法2：GDBを用いたソースコードデバッグ

ELRKではGDB環境も提供しているので，これを用い，ソースコードレベル・デバッグを行うことができる．GDB自体はLinux環境のみではなく，さまざまな環境で親しまれているデバッガなので，名前だけでも耳にした方は多いと思う．それでは，どのように使うのかを解説していく．

▶GDBを用いたソースコードデバッグ

STEP1：準備

ソースコードデバッグを行う場合，プログラムの構築（コンパイル）に関しても準備が必要である．具体的には，コンパイル時に専用オプション（-g）をつけることになる．つまり，コンパイル時に次のような操作となる．

```
> arm_720t_le-gcc -g program.c
```

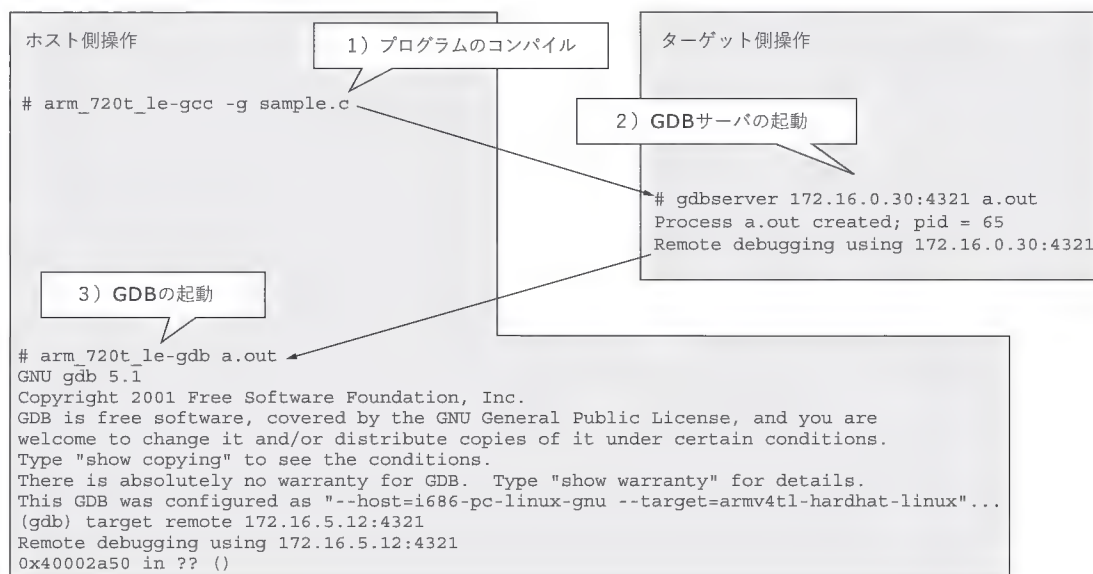
これにより，生成されたバイナリにデバッグ専用情報が付加され，結果としてソースコードデバッグができる．

STEP2：操作

それでは，実際に提供されているサンプルを用いてデバッグ操作を行ってみる．サンプルプログラムは，ホストコンピュータのターゲット用ルートディレクトリ（/opt/hardhat/refket/arm/720t_le/target）の下のrootディレクトリにある．このディレクトリ上で，クロスコンパイラを用いて，専用オプション付きでコンパイルする．コンパイルが終了したら，今度はターゲット上の操作で，できたバイナリをGDBサーバと一緒に起動する．起動が成功したら，またホストコンピュータの操作に戻り，GDB本体を立ち上げることで，デバッグ開始となる（**図5**）．

＊　＊

実際のデバッグ操作は，ホストコンピュータ上で行うことに

〔図5〕サンプルを使ったデバッグ操作

なる．たとえば，GDB上でlistと打つと，ソースコードの表示ができるし，break N(Nは行番号)と打てば，ブレーク設定ができる．プログラムを実行させたければ，contと入力するだけである．その後，ブレーク設定した場所までプログラムが動作すれば，その場でプログラムが停止(ブレークヒット)する．

ちなみにGDBサーバと一緒に起動したプログラムは，main()で停止しており，GDBから実行指示があるまで動かない．また，変数を表示したければ，print <変数名>と入力すれば，変数の内容が表示される．ただし注意が必要なのは，変数表示は，プログラムが停止している場合のみ使用可能だということである．したがって通常は，“ブレーク設定→実行→ブレークヒット→変数確認→新規ブレーク設定→実行……”，といった方法で，どこまでプログラムが実行したか，変数がどのようになっているか，を逐次確認しながらデバッグを行うと思うが，万が一，ブレーク設定を間違えて，プログラムが停止しない場合，GDB上で，^C(コントロールキーと，Cボタンを同時に押す)と入力するだけで，即座にプログラムを停止させることもできる．あとは，問題箇所を見つけて，修正，再実行させるだけである．付け加えると，再構築したプログラムをGDBでデバッグする場合，その都度，GDBサーバと一緒に起動する必要がある．

● 補足

-gオプションをつけた場合，バイナリサイズが肥大化するので，つねにオプション付加をすることはあまり得策ではない．今回使用したプログラムについて，オプションありと，なしの場合の容量を次に示す．

- オプションなし：5679バイト
- オプションあり：7535バイト

プログラムの内容がそれほど複雑ではないが，それでも4割近くサイズが増加する．したがって，デバッグ対象とするプログラム(ファイル)にのみオプションを適用させるのが，理想だといえる．

スタンドアロンへの道

さて，これでターゲットボード上で必要なアプリケーションも動作させることができ，サーバとしての機能は実現できたといえる．ただし，この状態だと，ファイルシステムがホストコンピュータのNFS公開ディレクトリを使用しているので，ファイルシステムを何らかの方法でターゲットボード上に単独で保有させることをしないかぎり，常にホストコンピュータが必要になってしまう．

● CFカードの動作

Armadilloには CFカードスロットがあり，ここにメモリ系CFカードを装着すれば，IDEドライブとして使用できる．また，CFカードは近年，大容量のものが比較的入手しやすいので，これをターゲットボード上のファイルシステムに使用することにする．Armadilloでこのスロットを使うためには，IDEドライブの設定がなされていれば，結果として使用可能となる．設定の確認は，xconfigでできる[注2](**図6**)．

● ファイルシステムの確定と保存

さて，今度はCFカードにファイルシステムを作成し，必要なファイルをコピーしなければならない．この作業は，ホストコンピュータで行うほうが簡単なので，次にホストコンピュータでの作業を説明する．ただし，ホストコンピュータでCFカードが読み書きできる環境が必要になる．実際には，CFカードをLinuxでよく使われるファイルシステムである“ext2”でフォーマットするだけである．ちなみにRed Hat Linuxの場

Column 2

デバッガと虫眼鏡/デバッグと殺虫剤

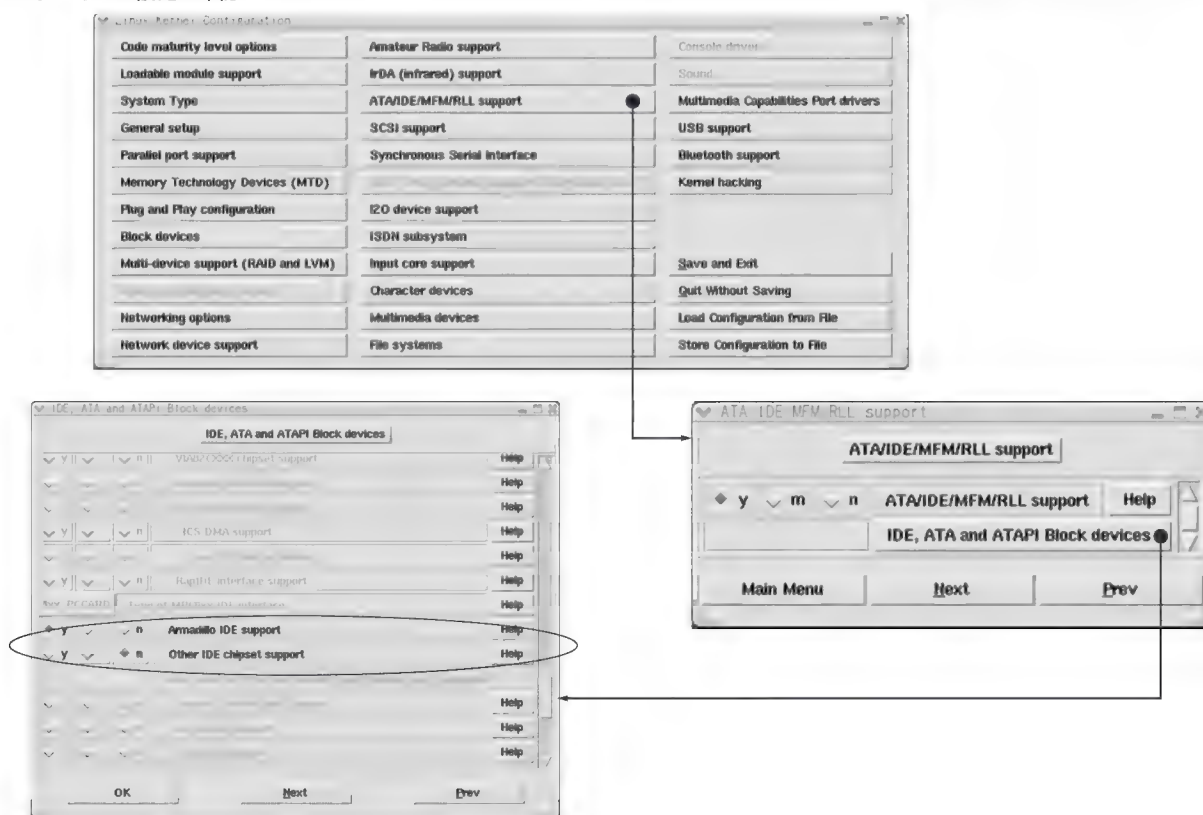
ちょっと，突拍子もない題目かもしれないが，デバッガに関してよく質問されることに，どのようにしてバグを発見するのか，どのようにしてバグを潰すのか，といった話がある．

正直なところ，バグ発見の方法については，これといった手順はなく，その人のスキルに依存しているとしかいえないのではないだろうか．このような質問をする人と話をしていると，どうもデバッガの機能にバグをつぶす機能が入っていると考え，デバッガを使えば，簡単(自動的に？)にバグを発見しつぶせると考えているように思えてならない．

ただ，よく考えてみてほしいのだが，デバッガはバグ発見の道具でしかなく，しいていえば“虫眼鏡”のようなものでしかない．つまり，虫眼鏡を使えば虫(バグ)を見つけることはできるが，虫眼鏡自体が虫を勝手に探してくれるわけではないし，もちろん，虫を殺す“殺虫剤”の機能があるわけではない．バグを殺す“殺虫剤”は，あくまでデバッグ作業そのものということになる．

また現時点では，ソフトウェアのバグだけに効き目がある「殺虫剤」は開発されていないので，虫眼鏡で地道にバグを見つけてそれに対してピンポイントでつぶすしかない．かりに殺虫剤があるとしたら，それは“ファイル消去”を意味してしまうのではないだろうか？　つまり，使った人まで殺すような強力な毒薬しかないということである．もしかしたら，遠い将来に，バグ潰しまでできるデバッガが生まれるかもしれないが，プログラムのバグはどうやって見分けるのだろうか．物理的な形をしていれば(ゴ○ブリみたい!?)わかるけれど……．

注2：今回は，Armadilloが“True-IDE”機能をサポートしており，CFカードスロットがそれ専用に設計されているため，このようなことができた．したがって，CFカードをもつターゲットボードがすべて，これと同じことができるとはかぎらないので注意が必要．

〔図6〕IDEドライブの設定の確認

合，コマンドとしてmkfs.ext2があるので，これを使用することをおすすめする．具体的には，パーティション設定を行い，このコマンドを実行するだけとなる．

ファイルシステム環境ができたら，必要なファイルをCFカードにコピーするのだが，コピーする内容は，ホストコンピュータにあるターゲット用ルートファイルシステム以下すべてが理想である．ただし注意が必要なのは，各種属性などの引き継ぎが必要なことである．筆者は，tarを用いて，コピーを行い，これを実現した．具体的には，tarコマンドを用いて，コピー元ディレクトリをすべて，一つのファイル(tarイメージ)とし，それをCFカードに展開するといった方法である．

● Linuxの起動設定

以上で，CFカードにファイルシステムを作ることができるので，あとはそれを装着し，Linuxの起動設定で，ファイルシステムの引用元をCFカードに指定するだけとなる．指定方法は，起動パラメータの変更となるので，変更時の内容のみを次に記す．

```
ip=XXX.XXX.XXX.XXX root=/dev/hda1
```

● RedBootの設定

あとは完成したLinuxイメージをターゲットボードに転送し，FISに保存し，ブートローダ(RedBoot)から自動起動できる構成とするだけで，電源ONと連動し，Linuxの起動ができるようになる．次に，RedBootの自動起動設定を記しておく．

```
Fis load linux
exec -b 0x28000 -l YYYYYYY 0xc0028000
```

おわりに

今回の作業を振り返り，注意しなければいけないこととして，

- 実装されているデバイスドライバの内容
- オープンソースのクロス開発

の2点がある．今回使用したELRK-Armadilloでは，その物理的な構造も手助けし，非常に簡単に目的物を構築することができたが，ELRKのすべてが同様ではなく，対象とするターゲットボードの構造などによって，同じことを行おうとしても，今回以上に作成しなければならないものがあることもある．とくに，デバイスドライバの内容はボード依存となるので注意してほしい．

わたなべ・たけお　(株)イーエルティィ

初級ドライバ開発者のための

Windowsデバイスドライバ開発テクニック

第2回　ドライバとアプリケーションの通信の方法

丸山治雄

前回は，DriverEntry()の詳細について解説しました．今回は，残りの基本関数，Unload関数，Create/Close関数について解説し，続いてドライバとアプリケーションの通信方法について解説します．

2.1 Unload関数の詳細

Unload関数(KIT1050Unload())は，デバイスドライバを停止してアンロードするときに呼び出されます(**リスト2.1**)．

処理する内容は，割り込みオブジェクトの削除，システムに登録したリソースの削除(Windows NTのとき)，ドライバ内で使用したPCIメモリまたはポートアドレスの削除，シンボリックリンクの削除，およびデバイスオブジェクトの削除を行います(①)．

なお，通常はデバイスクローズ(クローズディスパッチ)の処理で行いますが，念のためメモリマップも削除します．

2.2 Create/Close関数の詳細

Create/Close関数は，アプリケーションがドライバ(デバイス)をオープン/クローズしたときに呼び出される関数で，メモリマップの作成，KIT1050のレジスタ設定などを行います．ここでは，**リスト2.2**のように，KIT1050CreateClose()でメッセージを受け取り，処理要求によってCreate(IRP_MJ_CREATE)とClose(IRP_MJ_CLOSE)の実際の処理に振り分けています．

Create/Close関数は，処理する内容がなくても必ずメッセージを受け取り，返り値としてSTATUS_SUCCESSを返します．もし，処理する内容がないからメッセージを受け取らない，あるいはSTATUS_SUCCESSを返さないと，アプリケーション側のCreateFile()が処理に失敗し，ハンドル値はINVALID_HANDLE_VALUEとなってしまいます．

● IRP_MJ_CREATE(Create)

IRP_MJ_CREATEは，アプリケーションがCreateFile()でドライバ(デバイス)をオープンするときに必ずリクエストが来ます．

このドライバでは，KIT1050の初期化，そして割り込みを許可するためにPLX9054の割り込み許可ビットをイネーブルにしています．また，アプリケーションからPCIボードのメモリレジスタとSRAMメモリを直接アクセスするためのメモリマップを作成しています．

〔**リスト2.1**〕**Unload関数**(DriverEntry.c/KIT1050Unload()から)

```
VOID KIT1050Unload( IN PDRIVER_OBJECT DriverObject )
{
  UNICODE_STRING         deviceLinkUnicodeString;

  PDEVICE_EXTENSION pExtension =
       DriverObject->DeviceObject->DeviceExtension;
  PKIT1050_DRIVER_INFO PCIinf = pExtension->PCIinf;
  PKIT1050_REGISTER PCIRegPointer =
                 pExtension->PCIRegPointer;
  PPCI_PLX_CONFIG PCI9054RegPointer =
                 pExtension->PCI9054RegPointer;

  // 割り込みの解除                                              ┐
  if ( pExtension->InterruptObject != NULL )                      │
    IoDisconnectInterrupt(pExtension->InterruptObject);           │
                                                                  │
  // リソースの削除                                               │
  if ( Win2000 == 0 )                                             │
    PCIRemoveResource( DriverObject );                            │
                                                                  │
  // ドライバメモリの解除                                         │
  if ( PCIinf->PCIMemAddr != 0 )                                  │
    MmUnmapIoSpace( (PVOID)PCIinf->PCIMemAddr,                    │
                           PCIinf->MemSize );                     │
  if ( PCIinf->PLXAddr != 0 )                                     │
    MmUnmapIoSpace( (PVOID)PCIinf->PLXAddr,                       │
                           PCIinf->PLXMemSize );                  │
  PCIinf->PCIMemAddr = (ULONG)0;                                  │
  PCIinf->PLXAddr = (ULONG)0;                                     │
                                                                  ├←①
  // アプリケーションメモリの解除                                 │
  if ( PCIinf->VMemAddr != 0L )                                   │
    ZwUnmapViewOfSection((HANDLE)-1,                              │
                          (PULONG)PCIinf->VMemAddr );             │
                                                                  │
  if ( PCIinf->VPLXAddr != 0L )                                   │
    ZwUnmapViewOfSection((HANDLE)-1,                              │
                          (PULONG)PCIinf->VPLXAddr );             │
                                                                  │
  PCIinf->VMemAddr = (ULONG)0;                                    │
  PCIinf->VPLXAddr = (ULONG)0;                                    │
                                                                  │
  // Delete the symbolic link                                     │
  RtlInitUnicodeString (&deviceLinkUnicodeString,                 │
                        deviceLinkBuffer );                       │
  IoDeleteSymbolicLink (&deviceLinkUnicodeString);                │
                                                                  │
  // Delete the device object                                     │
  IoDeleteDevice (DriverObject->DeviceObject);                    ┘
}
```

〔リスト 2.2〕**Create/Close 関数**(DriverEntry.c/KIT1050CreateClose()から)

```
NTSTATUS KIT1050CreateClose(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )
{
  PDEVICE_EXTENSION pExtension =
            DeviceObject->DeviceExtension;
  PKIT1050_DRIVER_INFO PCIinf = pExtension->PCIinf;
  PKIT1050_REGISTER PCIRegPointer =
            pExtension->PCIRegPointer;
  PPCI_PLX_CONFIG PCI9054RegPointer =
           pExtension->PCI9054RegPointer;
  PIO_STACK_LOCATION irpStack;
  ULONG              inputBufferLength;
  ULONG              outputBufferLength;
  NTSTATUS  status = STATUS_SUCCESS;  // Driver status;

  // アプリケーションとの通信に使用する IRP のポインタを取り出す
  irpStack = IoGetCurrentIrpStackLocation(Irp);
  // ドライバがアプリケーションから受け取るデータサイズ
  inputBufferLength  =
      irpStack->Parameters.DeviceIoControl.InputBufferLength;
  // アプリケーションが受け取れるデータサイズ
  outputBufferLength =
      irpStack->Parameters.DeviceIoControl.OutputBufferLength;

  Irp->IoStatus.Status = STATUS_SUCCESS ;
  Irp->IoStatus.Information = 0 ;

  switch (irpStack->MajorFunction){
    case IRP_MJ_CREATE: {
      PHYSICAL_ADDRESS mema;
      PHYSICAL_ADDRESS memb;

      ULONG  Mem;
      ULONG  BusNm;

      // KIT1050 Board Reset                    ┐
      PCIRegPointer->CTRL_0 = CTRL0_BRST;       │
      // KIT1050 Int. Reset                     ├←①
      PCIRegPointer->CTRL_1 = 0;                │
      PCIRegPointer->LED = 0x0f;                ┘

      // 割り込みを有効にする
      // PLX9054 のマスタ割り込み解除
      // Shared Run Time Register (Local Interrupt Enable bit on)
      if ( EnableInt != 0 )
        PCI9054RegPointer->SHARED_ICS |= PCI90X0_LOCALINTENABLE; ←②

      BusNm = pExtension->DeviceNode.BusNumber;

      // User mmap                                                         ┐
      mema.LowPart = (ULONG)PCIinf->pPCIMemAddr;                           │
      mema.HighPart = 0L;                                                  │
      Mem = (ULONG)0x0000;                                                 │
      HalTranslateBusAddress( PCIBus, BusNm, mema, &Mem, &memb );          ├③
      PCIinf->VMemAddr = (ULONG)MapMemMapTheMemory (                       │
                            &memb, PCIinf->MemSize );                      │
      PCIinf->VMemSize = PCIinf->MemSize;                                  ┘

      mema.LowPart = (ULONG)PCIinf->pPLXAddr;                              ┐
      mema.HighPart = 0L;                                                  │
      Mem = (ULONG)0x0000;                                                 │
      HalTranslateBusAddress( PCIBus, BusNm, mema, &Mem, &memb );          ├④
      PCIinf->VPLXAddr = (ULONG)MapMemMapTheMemory (                       │
                            &memb, PCIinf->PLXMemSize );                   │
      PCIinf->VPLXMemSize = PCIinf->PLXMemSize;                            ┘
    } break;

    case IRP_MJ_CLOSE: {
      pExtension->nCount = 0;
      // 割り込みを無効にする
      // PLX9054 のマスタ割り込み禁止
      // Shared Run Time Register (Local Interrupt Enable bit off)
      if ( EnableInt != 0 )
        PCI9054RegPointer->SHARED_ICS &= ~PCI90X0_LOCALINTENABLE; ⑤

      // ユーザー用メモリマップの解除
      if ( PCIinf->VMemAddr != 0L ) {                                      ┐
        ZwUnmapViewOfSection((HANDLE)-1,                                   │
                              (PULONG)PCIinf->VMemAddr );                  │
        PCIinf->VMemAddr = (ULONG)0;                                       │
      }                                                                    ├⑥
      if ( PCIinf->VPLXAddr != 0L ) {                                      │
        ZwUnmapViewOfSection((HANDLE)-1,                                   │
                              (PULONG)PCIinf->VPLXAddr );                  │
        PCIinf->VPLXAddr = (ULONG)0;                                       │
      }                                                                    ┘
    } break;
  }

  IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
  return( status );
}
```

具体的には，①では，**リスト 1.9**(第1回掲載)で取得したメモリレジスタのアドレスを使用してボードのリセットを行い，オープンが成功したことを示すためにLEDをすべて消灯しています．

②では，レジストリパラメータで割り込みを使用するときに，PLX9054の割り込み許可ビットをイネーブルにします．

③では，アプリケーションからメモリレジスタとSRAMを直接アクセスできるようにメモリマップを作成しています．VmemAddrの内容をアプリケーションでメモリの先頭としてアクセスすると，ドライバを介さずに直接アクセスできます．

同様に④で，PLX9054ローカルコンフィグレーションレジスタをアプリケーションから直接アクセスできるように処理しています．

● IRP_MJ_CLOSE(Close)

IRP_MJ_CLOSEは，アプリケーションがCloseHandle()でドライバ(デバイス)をクローズしたときに，必ずリクエストが来ます．アプリケーションが異常終了してCloseHandle()を呼び出さないときもリクエストが来ます．

基本的には，IRP_MJ_CREATEにより作成したメモリマップを削除し，割り込み禁止などの処理を行います．

具体的には，⑤でPLX9054の割り込み許可ビットをクリアしてKIT1050からの割り込み発生を禁止しています．⑥で，アプリケーションに解放したKIT1050のメモリとPLX9054のメモリレジスタのポインタを解放しています．

2.3 MapMemMapTheMemory()の詳細

MapMemMapTheMemory()は，アプリケーションからPCIボードのメモリを直接アクセスできるように，物理アドレスから仮想アドレスに変換します(**リスト 2.3**)．この関数に渡す物理アドレスは，事前にHalTranslateBusAddress()によりシステム論理アドレスに変換しておく必要があります．

〔リスト2.3〕物理アドレスから仮想アドレスに変換(MEMORYMP.C/MapMemMapTheMemory()から)

```
⑤ PULONG MapMemMapTheMemory(
     IN PHYSICAL_ADDRESS   *physicalAddress,
     IN ULONG MemSize
 )
{
  ULONG              length;
  UNICODE_STRING     physicalMemoryUnicodeString;
  OBJECT_ATTRIBUTES objectAttributes;
  HANDLE        physicalMemoryHandle  = NULL;
  PVOID         PhysicalMemorySection = NULL;
  NTSTATUS      ntStatus;
  PHYSICAL_ADDRESS physicalAddressBase = *physicalAddress;
  PHYSICAL_ADDRESS viewBase;
  PVOID         virtualAddress;


  // メモリマップ作成用のオブジェクト名                              ┐
  RtlInitUnicodeString (&physicalMemoryUnicodeString,              ├←①
              L"¥¥Device¥¥PhysicalMemory");                        ┘

  // Zwxxxxx()関数用OBJECT_ATTRIBUTES構造体を初期化                 ┐
  InitializeObjectAttributes (&objectAttributes,                    │
                   &physicalMemoryUnicodeString,                    ├←②
                   OBJ_CASE_INSENSITIVE,                            │
                   (HANDLE) NULL,                                   │
                   (PSECURITY_DESCRIPTOR) NULL);                    ┘

  // メモリマップ作成用のハンドルを取得                              ┐
  ntStatus = ZwOpenSection (&physicalMemoryHandle,                  ├←③
                   SECTION_ALL_ACCESS,                              │
                   &objectAttributes);                              ┘

  if (!NT_SUCCESS(ntStatus)) {
    ntStatus = -1;
    goto done;
  }

  // オブジェクトハンドルのロック                                    ┐
  ntStatus = ObReferenceObjectByHandle (                            │
             physicalMemoryHandle,                                  ├←④
             SECTION_ALL_ACCESS,                                    │
             (POBJECT_TYPE) NULL,                                   │
             KernelMode,                                            │
             &PhysicalMemorySection,                                │
             (POBJECT_HANDLE_INFORMATION) NULL);                    ┘

  if (!NT_SUCCESS(ntStatus)) {
    ntStatus = -1;
    goto close_handle;
  }

  // メモリマップする、物理アドレスとサイズ                          ┐
  length = MemSize;                                                 │
  viewBase = *physicalAddress;                                      │
  virtualAddress = NULL;                                            │
                                                                    │
  // 仮想メモリポインタへの変換                                      │
  ntStatus = ZwMapViewOfSection (physicalMemoryHandle,              │
                   (HANDLE) -1,                                     ├←⑤
                   &virtualAddress,                                 │
                   0L,                                              │
                   length,                                          │
                   &viewBase,                                       │
                   &length,                                         │
                   ViewShare,                                       │
                   0,                                               │
                   PAGE_READWRITE | PAGE_NOCACHE);                  ┘

  if (!NT_SUCCESS(ntStatus)) {
    ntStatus = -1;
    goto close_handle;
  }

  // メモリマップアドレスの調整
  (ULONG)virtualAddress +=
     (ULONG)physicalAddressBase.LowPart - (ULONG)viewBase.LowPart;
  ntStatus = (ULONG)virtualAddress;

close_handle:
  ZwClose (physicalMemoryHandle);

done:
  return( (PULONG)ntStatus );

}
```

①のRtlInitUnicodeString()で，仮想メモリマップ作成用のオブジェクトを作成します．オブジェクト名は，¥¥Device¥¥PhysicalMemoryに固定です．

②では，仮想メモリをロックするために使用するOBJECT_ATTRIBUTES構造体を初期化しています．

③でオブジェクトのハンドルを取得します．このハンドルを④でロックしています．⑤で物理アドレスから仮想アドレスに変換します．関数の返り値が-1のときは，メモリマップの作成に失敗したことを表します．

2.4 ドライバとアプリケーションの通信の方法

本連載で使用するソースファイルの一覧表(第1回の**表1.1**)に不足があったので，**表2.1**に不足分を示します．

アプリケーションからドライバにリクエストを送るときは，DeviceIoControl()を使用します．この関数を使用するうえで注意すべき点は，1回のリクエストで通信できるサイズが4096バイト以内であること，そして通信に時間がかかることです．

したがって，PCIボードのメモリにアクセスしたり大容量のデータをアクセスするときは，メモリマッピングやReadFile

〔表2.1〕デバイスドライバのソースファイル(表1.1の不足分)

ファイル	内容	
PCIRESOU.H	リソースをシステムに登録する構造体の定義．Windows NTのときにのみ使用する	
PCIDEF.H	PCIボードアクセス用関数を定義	
TYPEDEF.H	変数の型定義．Visual C++ 6.0では，とくに使用する必要はない	
KIT1050DAT.C	ドライバ内で使用する，変数の定義と初期化を行うソースファイル．実行されるモジュールはない	
INTSERVI.C	割り込み処理と割り込み後処理を行う関数を記述	
	KIT1050Interrupt Service()	PCIボードからの割り込みの処理
	KIT1050Dpc Routine()	割り込み後処理．アプリケーションに割り込みが発生したことを通知する
PCIINTER.C	割り込み登録処理とアプリケーションが取り消されたとき，通知登録のキャンセルを行う関数が記述されている	
	KIT1050Interrupt()	PCIボードからの割り込みを処理するモジュールを登録
	KIT1050IntCancel()	登録されている，アプリケーションからの通知依頼をリストから削除する
PCIIOCT.C	ユーザー定義コードのメッセージ処理を記述	
	KIT1050Device Control()	ユーザー定義コードのメッセージ処理

〔リスト 2.4〕WINIOCTL.H で定義されている IOCTL コード定義マクロ

```
#define CTL_CODE( DeviceType, Function, Method, Access )   ¥
( ((DeviceType) << 16) | ((Access) << 14) | ((Function) << 2) | (Method) )
```

〔リスト 2.5〕IOCTL 用のファンクションコード(APIDef.h から)

```
#define  USERIOCODE           0x0800  /* ユーザーコード */
#define  PCI_BOARDNUMBER        0x42  // PCI ボード数の取得
#define  PCI_GETCONFIG          0x43  // PCI Config Reg の取得
#define  PCI_REGISTERREAD       0x46  // ポートレジスタからの読み出し
#define  PCI_REGISTERWRITE      0x47  // ポートレジスタへの書き込み
#define  PCI_VERSION            0x48  // ドライバのバージョン
#define  PCI_PHYSMAP            0x4B  // PCI 物理メモリアドレスの取得
#define  PCI_MMAP               0x4C  // mmap() 論理ポインタ取得
#define  PCI_MEMLENGTH          0x4D  // メモリ・ポートのサイズ取得
#define  PCI_DEBUG              0x7F  /* ドライバのデバッグ用 */
```

〔リスト 2.6〕メッセージの分類
(DriverEntry.c/KIT1050Dispatch() から)

```
NTSTATUS KIT1050Dispatch(
  IN PDEVICE_OBJECT DeviceObject,
  IN PIRP Irp
  )
{
  PDEVICE_EXTENSION  pExtension =
                             DeviceObject->DeviceExtension;
  PLIST_ENTRY        head;
  PIRP               NewIrp;
  PIO_STACK_LOCATION irpStack;
  KIRQL              kCancelSpin;
  NTSTATUS           Status ;
  ULONG              ioControlCode;
  PULONG             IoBuffer;
  LONG               InputSize;
  LONG               OutputSize;

  // Init to default settings- we only expect 1 type of
  //   IOCTL to roll through here, all others an error.
  Irp->IoStatus.Status     = STATUS_SUCCESS;
  Irp->IoStatus.Information = 0;

  irpStack = IoGetCurrentIrpStackLocation(Irp);
  InputSize = irpStack->
                Parameters.DeviceIoControl.InputBufferLength ;
  OutputSize = irpStack->
               Parameters.DeviceIoControl.OutputBufferLength ;
  IoBuffer = (PULONG)Irp->AssociatedIrp.SystemBuffer ;

  switch (irpStack->MajorFunction) {
    // 一般 IOCTL 要求
    case IRP_MJ_DEVICE_CONTROL:
      ioControlCode = irpStack->
                        Parameters.DeviceIoControl.IoControlCode;
      // KIT1050_PCI vendor Command
      if ( (ioControlCode & (USERIOCODE << 16) ) ==
                                       (USERIOCODE << 16) )
        return( KIT1050DeviceControl( DeviceObject, Irp ) );
  }
```

/WriteFile 関数を使用し，I/O マネージャとの通信をなるべく使用しない方法でアクセスすることを推奨します注2.1.

2.5 ユーザー定義のディスパッチ定義

アプリケーションがドライバと通信するためには，IOCTL 用のファンクションコードを定義する必要があります．

注 2.1 ：PCI ボードのメモリにアクセスする方法については別途説明する．

ファンクションコードは，**リスト 2.4** のように定義することが規定されています．

DeviceType はユーザー定義の場合，0x0800 から使用するように規定されています．Function は任意のユーザーコードを定義します(**リスト 2.5** 参照)．

Method と Access を設定していませんが，METHOD_BUFFERED と FILE_ANY_ACCESS はともに 0 なので省略しています．

2.6 ユーザー定義のディスパッチ処理

ユーザー定義のディスパッチ処理を行うときは，DriverEntry() で，

```
DriverObject->
   MajorFunction[IRP_MJ_DEVICE_CONTROL]
      = KIT1050Dispatch;
```

を定義して，ディスパッチ処理を I/O マネージャに登録する必要があります(**リスト 1.11** 参照．第 1 回掲載)これにより，ユーザーからリクエストが来たときに KIT1050Dispatch() にメッセージが渡されます．

● メッセージの分類(**リスト 2.6**)

ユーザー定義のメッセージは，MajorFunction が IRP_MJ_DEVICE_CONTROL として分類されてきます．この中で，さらに Parameters.DeviceIoControl.IoControlCode にユーザー定義コードがセットされているので，各コードにより処理を行います．

今回のサンプルでは，ユーザー定義コードのメッセージは次項で説明する KIT1050DeviceControl() で処理しています．KIT1050Dispatch() 内では，次回以降で説明する DMA テーブルの作成などのディスパッチ処理を行っています．

● ユーザー定義コードのメッセージ処理の詳細(**リスト 2.7**)

ユーザー定義コードのメッセージ処理を**リスト 2.7** に，アプリケーションからドライバへのリクエスト例を**リスト 2.8** に示します．

● IOCTL_PCI_BOARDNUMBER(ボード数の取得)

FindKIT1050_PCI() で検索した PCI ボードの枚数をアプリケーションに通知します(①)．

アプリケーションからリクエストを発行するとき，アプリケーションからドライバに渡すデータはないので，アドレスは NULL，サイズは 0 にしてあります(**リスト 2.8** の①)．

● IOCTL_PCI_MMAP(PCI メモリポインタの取得)

アプリケーションから PCI メモリを直接アクセスするために

〔リスト2.7〕ユーザー定義コードのメッセージ処理(PCIIOCT.C/KIT1050DeviceControl()から)

```
#include <stdio.h>
#include "ntddk.h"
#include "driverdef.h"

NTSTATUS KIT1050DeviceControl(
  PDEVICE_OBJECT DeviceObject,
  PIRP Irp
  )
{
  PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackLocation(Irp);
  PDriverRequest  Request;
  Driver_Open  *DRVHdl= (PVOID)Irp->AssociatedIrp.SystemBuffer;
  PDEVICE_EXTENSION  pExtension = DeviceObject->DeviceExtension;
  PKIT1050_DRIVER_INFO  PCIinf = pExtension->PCIinf;
  NTSTATUS  status = STATUS_SUCCESS;  // Driver status;

  ULONG  Handle = (ULONG)-1;
  ULONG  IoControlCode;
  ULONG  rc = (ULONG)-1L;        // Function status;

  LONG  OutputBufferLength;
  LONG  InputBufferLength;

  Request = (PDriverRequest)Irp->UserBuffer;
  IoControlCode = irpStack->Parameters.DeviceIoControl.IoControlCode;

  OutputBufferLength =
     irpStack->Parameters.DeviceIoControl.OutputBufferLength;
  InputBufferLength =
     irpStack->Parameters.DeviceIoControl.InputBufferLength;
  Irp->IoStatus.Information = 0;

  switch( IoControlCode ) {
// PCIボード数の取得
    case IOCTL_PCI_BOARDNUMBER: {
      PULONG  BoardCnt = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      *BoardCnt = PCI_Board_Number;
      Irp->IoStatus.Information = sizeof(ULONG);
    } break;

// PCIメモリポインタの取得
    case IOCTL_PCI_MMAP: {
      PPCIMMAP  DRVmap = (PPCIMMAP)Irp->AssociatedIrp.SystemBuffer;
      PULONG  BoardNum = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      PULONG  Vmem = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      Board = *BoardNum;  // 未使用
      *Vmem++ = (ULONG)PCIinf->VMemAddr + KIT1050REGISTER_OFFSET;
      *Vmem++ = (ULONG)PCIinf->VMemAddr + KIT1050MEMORY_OFFSET;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = (ULONG)PCIinf->VPLXAddr;
      *Vmem++ = 0;
      Irp->IoStatus.Information = sizeof(ULONG) * 8;
    } break;

// PCI 物理メモリアドレスの取得
    case IOCTL_PCI_PHYSMAP: {
      PPCIMMAP  DRVmap = (PPCIMMAP)Irp->AssociatedIrp.SystemBuffer;
      PULONG  Vmem = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      *Vmem++ = (ULONG)PCIinf->pPCIMemAddr;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = (ULONG)PCIinf->pPLXAddr;
      *Vmem++ = (ULONG)PCIinf->pPLXioAddr;
      Irp->IoStatus.Information = sizeof(ULONG) * 8;
    } break;

// PCI メモリサイズの取得
    case IOCTL_PCI_MEMLENGTH: {
      PPCIMMAP  DRVmap = (PPCIMMAP)Irp->AssociatedIrp.SystemBuffer;
      PULONG  Vmem = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      *Vmem++ = KIT1050REGISTER_SIZE;
      *Vmem++ = (ULONG)PCIinf->MemSize;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = 0;
      *Vmem++ = (ULONG)PCIinf->PLXMemSize;
      *Vmem++ = (ULONG)PCIinf->PLXioSize;
      Irp->IoStatus.Information = sizeof(ULONG) * 8;
    } break;

// Port register read
    case IOCTL_PCI_REGISTERREAD: {
      PPCIREGISTERREAD  RegRead =
        (PPCIREGISTERREAD)Irp->AssociatedIrp.SystemBuffer;
      PULONG  RegAddr;
      LONG  Access;
      ULONG  Value;

      Access = RegRead->Access;
      if ( RegRead->RegisterMode == 0 )
      {       // Mem register
        RegAddr = (PULONG)(PCIinf->PCIRegAddr +
                   RegRead->Register);
        if ( Access == IOREAD_BYTE )
          Value = *(PUCHAR)RegAddr;
        else if ( Access == IOREAD_SHORT )
          Value = *(PUSHORT)RegAddr;
        else
          Value = *RegAddr;
      } else {       // PLX Port
        RegAddr = (PULONG)(PCIinf->PLXioAddr +
                     RegRead->Register);
        if ( Access == IOREAD_BYTE )
          Value = READ_PORT_UCHAR( (PUCHAR)RegAddr );
        else if ( Access == IOREAD_SHORT )
          Value = READ_PORT_USHORT( (PUSHORT)RegAddr );
        else
          Value = READ_PORT_ULONG( RegAddr );
      }
        RegRead->Value = Value;
          Irp->IoStatus.Information = sizeof(
                                      PCIREGISTERREAD );
    } break;

// Port register write
    case IOCTL_PCI_REGISTERWRITE: {
      PPCIREGISTERWRITE  RegWrite =
       (PPCIREGISTERWRITE)Irp->AssociatedIrp.SystemBuffer;
      PULONG  RegAddr;
      LONG  Access;
      ULONG  Value;

      Access = RegWrite->Access;
      Value = RegWrite->Value;
      if ( RegWrite->RegisterMode == 0 )
      {       // Mem register
        RegAddr = (PULONG)(PCIinf->PCIRegAddr +
                   RegWrite->Register);
        if ( Access == IOREAD_BYTE )
          *(PUCHAR)RegAddr = (UCHAR)Value;
        else if ( Access == IOREAD_SHORT )
          *(PUSHORT)RegAddr = (USHORT)Value;
        else
          *RegAddr = Value;
      } else {       // PLX Port
        RegAddr = (PULONG)(PCIinf->PLXioAddr +
                     RegWrite->Register);
        if ( Access == IOREAD_BYTE )
          *(PUCHAR)RegAddr = (UCHAR)Value;
        else if ( Access == IOREAD_SHORT )
          *(PUSHORT)RegAddr = (USHORT)Value;
        else
          *RegAddr = Value;
      }
        Irp->IoStatus.Information = 0;
    } break;

// PCI Configの読み出し
    case IOCTL_PCI_GETCONFIG: {
      PPCIGETCONFIG  DRVConfig =
            (PVOID)Irp->AssociatedIrp.SystemBuffer;
      RtlCopyMemory(Irp->AssociatedIrp.SystemBuffer,
            &pExtension->ConfigReg,
            sizeof(PCI_COMMON_CONFIG) );
      Irp->IoStatus.Information = sizeof(PCI_COMMON_CONFIG);
      break;
    } break;
```

〔リスト 2.7〕ユーザー定義コードのメッセージ処理(PCIIOCT.C/KIT1050DeviceControl()から)(つづき)

```
// ドライババージョン
    case IOCTL_PCI_VERSION: {
      PULONG  Ptr = (PULONG)Irp->AssociatedIrp.SystemBuffer;
      *Ptr = 0x00010000;    // Hi=Major Lo=Minor                    ⑦
      Irp->IoStatus.Information = sizeof( LONG );
    } break;

// デバッグ
    case IOCTL_PCI_DEBUG: {
      PDriverRequest Debug =
          (PVOID)Irp->AssociatedIrp.SystemBuffer;
      PULONG  APmem = (PVOID)Irp->AssociatedIrp.SystemBuffer;
      LONG  Category = Debug->Category;
      LONG  Function = Debug->Function;
      PULONG  Request;
      Request = (PULONG)&Debug->Request;
                                                                     ⑧
      switch ( Category ) {
        case 0:
          Irp->IoStatus.Information = 0;
          break;
        case 1:
          Irp->IoStatus.Information = 0;
          break;
        default:
          status = STATUS_INVALID_PARAMETER;
          break;
      }

    } break;

    default:
      status = STATUS_INVALID_PARAMETER;
      break;
  }
  Irp->IoStatus.Status = status;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
  return(status);
}
```

〔リスト 2.8〕アプリケーションからドライバへのリクエスト例

```
LONG   PCIBoardNumber = 0;
ULONG  PCIPointer[8] = { 0, 0, 0, 0, 0, 0, 0, (ULONG)-1};
ULONG  cbBytesReturned;
LONG   PciCnt = 0;

//  PCI ボード数の取得
DeviceIoControl(hPCIHdl, IOCTL_PCI_BOARDNUMBER,
        (LPVOID)NULL, 0,                                  ←①
        (LPVOID)&PCIBoardNumber, sizeof( LONG ),
        &cbBytesReturned, NULL);

if ( PCIBoardNumber == 0 ) {
  CloseHandle(hPCIHdl);
  hPCIHdl = NULL;
  return( NULL );
}

// PCIメモリポインタの取得
DeviceIoControl(hPCIHdl, IOCTL_PCI_MMAP,
        (LPVOID)&PciCnt, sizeof( LONG ),                  ←②
        (LPVOID)&PCIPointer[0], sizeof(LONG) * 8,
        &cbBytesReturned, NULL);
```

必要となるメモリポインタをアプリケーションに通知します．

アプリケーションからリクエストを発行するとき，アプリケーションからドライバに渡すデータはリクエストするボードの番号です(複数枚のとき．今回は未使用．**リスト 2.8**の②)．

ドライバは，8個のポインタをアプリケーションに通知します．[0]はレジスタメモリの先頭ポインタ，[1]はSRAMメモリの先頭ポインタ，[6]はPLX9054のローカルコンフィグレーションレジスタの先頭ポインタが格納されます．この値をPULONG(unsigned long *)の変数に置き換えて，通常のポインタ参照を行えばPCIメモリにアクセスできます．

ここで，PCIメモリをアクセスするうえで必要となる注意点をまとめます．

1. PCIのメモリアクセスは，32ビットのロングワードアクセスと規定されていますが，バイト，またはワードアクセスも可能な形でハードウェアを設計してください．これは，誤ってバイトまたはワードアクセスしたときの誤動作を防ぐためで，この設計がなされていなかったためにソフトの不具合がなかなか見つからなかったことがありました．
2. Watcom Cコンパイラを使用するときは，PULONGでアクセスしているつもりがバイトアクセスに変換されてしまう場合があるので注意してください．
   たとえば，
   ```
   PULONG Mem1;
   *Mem1 = 1;
   ```
   と記述すると，ダブルワードアクセスではなくバイトアクセスとしてコンパイルされます．これを防ぐためには，定数も一度変数に割り当ててから使用してください．他のコンパイラではこのような現象は確認していません．

● IOCTL_PCI_PHYSMAP(PCI物理アドレスの取得)

アプリケーションでは使用しません．

● IOCTL_PCI_MEMLENGTH(PCIメモリサイズの取得)

各メモリの有効サイズを通知します．ここで通知するサイズはドライバが認識するサイズではなく，メモリレジスタの有効サイズ，SRAMの有効サイズになります(③)．

● IOCTL_PCI_REGISTERREAD(PLX9054ローカルコンフィグレーションレジスタの読み取り)

PLX9054ローカルコンフィグレーションレジスタの内容をポートから読み取ります．

ローカルコンフィグレーションレジスタの内容は，メモリレジスタからアクセスできるので，実際にポートアクセスを使用することはありません(④)．

● IOCTL_PCI_REGISTERWRITE(PLX9054ローカルコンフィグレーションレジスタへの書き込み)

PLX9054ローカルコンフィグレーションレジスタへポートから書き込みます．

ローカルコンフィグレーションレジスタの内容は，メモリレジスタからアクセスすることができるので，実際にポートアクセスを使用することはありません．ポートアクセスのサンプルとして参考にしてください(⑤)．

● IOCTL_PCI_GETCONFIG(PCIコンフィグレーションレジス

タの取得)

FindKIT1050_PCI()で取得したPCIコンフィグレーションレジスタの内容をアプリケーションに通知します．PCIコンフィグレーションレジスタの構造体の定義はVisual C++にはないので，独自に定義しています(⑥)．

● IOCTL_PCI_VERSION(ドライババージョンの取得)

ドライバのバージョンをアプリケーションに通知します．アプリケーションのバージョンの相違から来る問題を回避するため，バージョンチェックを入れておくことを推奨します(⑦)．

● IOCTL_PCI_DEBUG(ドライバのデバッグ用)

ドライバの簡単なデバッグを行うとき(変数の確認など)に準備すると便利です．WinDbgやSoftICEを使用するほどでもないときにこのディスパッチメッセージを使用すると便利です．Categoryに目的別の値を設定して，Category別にアプリケーションに通知する変数を記述しておきます．アプリケーションの必要なところでドライバ内の変数をモニタします(⑧)．

2.7 割り込み処理ルーチンの登録

割り込み処理ルーチンはどのレベルでも登録できますが，一般的にはDriverEntry()かDriverObject->MajorFunction[IRP_MJ_CREATE]で指定したディスパッチ関数で登録するのが一般的です．

割り込み処理ルーチンを登録するにあたり，次の注意が必要です．

1. Windows NTではIRQリソースは共有できないと考えてください．たとえば，SCSIドライバやビデオドライバは他のIRQと競合して動作することを拒否します(占有)．
   したがって，その後に登録する優先順位の低いドライバは，たとえ共有で登録しようとしても拒否され，登録されないことがあります．このとき，IRQに空き番号があれば強制的に変更するか，割り込み処理をあきらめるしかありません．
2. Windows 2000/XPでは，物理IRQ番号と仮想IRQ番号が必ずしも一致するとは限りません．したがって，必ずHalAssignSlotResources()で取得した仮想IRQを使用します(**リスト1.9**参照．第1回掲載)．
3. IRQが他のデバイスと共有されているときは，割り込み処理ルーチンがデバイスに対する具体的な処理を組み込んでいないときでも，**リスト2.9**に示す処理だけは必ず組み込んでください．
   これは，自身が管理するデバイスが割り込みを発生させなくても，共有する他のデバイスが割り込み処理を要求してくることがあるからで，割り込み要因を調べて自身の割り込みでないときは**リスト2.9**のようにreturn(FALSE)で割り込み処理ルーチンから戻る必要があります(①)．

割り込み処理ルーチンは，次のように登録します．プログラムリストを**リスト2.10**に示します．なお，割り込み処理とは直接関係ありませんが，WaitQueueとQueueSpinを初期化しています．詳細は次回に説明します．

1. HalGetInterruptVector()により，割り込みベクタの取得を行います．このときのパラメータは次のようになります(①)．InterfaceTypeはPCIBusを指定します．BusNumberは，PCIボードを検出したバス番号(FindKIT1050_PCI()内で取得)を指定します．LevelとVectorは，同様にFindKIT1050_PCI()で取得した値をそれぞれ使用します．Irqlとaffinityは関数がセットするので，この値をそのままIoConnectInterrupt()に渡します．
2. 割り込みベクタが取得できたら，次に割り込み処理ルーチンを登録します(②)．IoConnectInterrupt()に引き渡すパラメータのうちInterruptModeはLevelSensitiveを設定してください．ShareVectorは基本的にはTRUE(共有)にしてください．占有にすると，共有しているデバイスがあったときに，割り込みが登録されないことがあります．
3. 最後に，割り込みが発生したとき後処理を行う関数を登録します(③)．これは必須ではありませんが，割り込み処理ルーチンは一般的に他のドライバなどの動作を停止して最優先に処理されます．このため，割り込み処理ルーチンでは最小限の処理を行い，時間のかかる処理は後で行います．今回は，プッシュボタンからの割り込みをアプリケーションに通知するので，DPC(Deferred Procedure Call)ルーチンを登録します．KIT1050DpcRoutine()はこのための処理です．IoInitializeDpcRequest()でDPCルーチンを登録します．DeviceObjectで設定されたパラメータがDPCルーチンのDeferredContextに渡されます．

〔リスト2.9〕必ず組み込むべき割り込み処理ルーチン

```
BOOLEAN KIT1050InterruptService(
      IN  PKINTERRUPT Interrupt,
      IN OUT PVOID Context
    )
{
  PDEVICE_OBJECT pDeviceObject = Context;
  PDEVICE_EXTENSION  pExtension =
                       pDeviceObject->DeviceExtension;
  PKIT1050_REGISTER  PCIRegPointer =
                       pExtension->PCIRegPointer;
  PPCI_PLX_CONFIG    PCI9054RegPointer =
                       pExtension->PCI9054RegPointer;

// 割り込み確認
  if ( (PCIRegPointer->STATUS & STS_INT) == 0 ) {
    // 他のデバイスの割り込みのときはFALSEを返す
    return( FALSE ); ←――――――――――――――――① 
  }
  // KIT1050からの割り込みのときはTRUEを返す
  // 割り込み処理
  return( TRUE );
}
```

〔リスト 2.10〕割り込み処理ルーチンの登録(PCIINTER.C/KIT1050Interrupt())

```
#include "ntddk.h"
#include "driverdef.h"
#include "pciresou.h"

extern  PCI_RESOURCE PCIResources[5];
extern  ULONG  IntExclusive;

VOID KIT1050DpcRoutine(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
    );

LONG KIT1050Interrupt( IN PDEVICE_OBJECT DeviceObject )
{
  ULONG       interruptVector;
  KIRQL       irql;
  KAFFINITY   affinity;
  NTSTATUS    status;
  PDEVICE_EXTENSION  pExtension = DeviceObject->DeviceExtension;
  BOOLEAN    IntShare;


  if ( PCI_Board_Number == 0 )
    return( STATUS_INSUFFICIENT_RESOURCES );

  // interruptVectorの取得
  interruptVector = HalGetInterruptVector( PCIBus,                  ┐
                      pExtension->BusNumber,                         │
                      pExtension->InterruptRes.Level,                ├←①
                      pExtension->InterruptRes.Vector,               │
                      &irql, &affinity);                             ┘
  if ( interruptVector == 0 )
    return( STATUS_INSUFFICIENT_RESOURCES );

  // 割り込みレベル
  pExtension->InterruptMode = LevelSensitive;

  // 割り込み処理ルーチンの登録
  if ( IntExclusive == 0 )                                           ┐
    IntShare = FALSE;  // 占有                                       │
  else                                                               │
    IntShare = TRUE;   // 共用                                       │
  status = IoConnectInterrupt(&pExtension->InterruptObject,          │
              KIT1050InterruptService,                               ├←②
              DeviceObject,                                          │
              NULL,                                                  │
              interruptVector, irql, irql,                           │
              pExtension->InterruptMode,                             │
              IntShare,                                              │
              affinity, FALSE);                                      ┘

  if( !NT_SUCCESS( status ) )
    return( status );

  // 割り込み、後処理ルーチンの登録
  IoInitializeDpcRequest( DeviceObject, KIT1050DpcRoutine ); ←③

  // WindowsNTのとき、リソース登録用
  PCIResources[4].Length = pExtension->InterruptRes.Vector;
  PCIResources[4].PhysicalAddress.LowPart =
PCIResources[4].Length;

  return( STATUS_SUCCESS );

}
```

〔リスト 2.11〕割り込み処理ルーチン(INTSERVI.C/KIT1050InterruptService())

```
#include "ntddk.h"
#include "Driverdef.h"

BOOLEAN KIT1050InterruptService(
      IN  PKINTERRUPT Interrupt,
      IN OUT PVOID Context
    )
{
  PDEVICE_OBJECT pDeviceObject = Context;
  PDEVICE_EXTENSION  pExtension =
                          pDeviceObject->DeviceExtension;
  ULONG  StsReg;
  ULONG  ICSReg;
  LONG  DMAInt = -1;

  PCIRegPointer = pExtension->PCIRegPointer;
  PCI9054RegPointer = pExtension->PCI9054RegPointer;
  StsReg = PCIRegPointer->STATUS;

// 割り込み確認
  if ( (StsReg & STS_INT) == 0 ) {
    // KIT1050のプッシュSW割り込みでは無い
    ICSReg = PCI9054RegPointer->SHARED_ICS;
    // KIT1050のDMA転送の終了を確認
    if ( (ICSReg & PCI90XX_DMA0_INTACTIVE) != 0 ) {
      (UCHAR)PCI9054RegPointer->DMA0_COMMAND_REG |=
                                       PCI90XX_DMA_CLEARINT;
      DMAInt = 0;
    } else if ( (ICSReg & PCI90XX_DMA1_INTACTIVE) != 0 ) {
      (UCHAR)PCI9054RegPointer->DMA1_COMMAND_REG |=
                                       PCI90XX_DMA_CLEARINT;
      DMAInt = 1;
    } else {
      // 他のデバイスからの割り込みのときはFALSEを返す
      return( FALSE );
    }

// DMA転送終了処理
    PCIRegPointer->LED = 2;  //DMA Int.
    return( TRUE );
  }

  // 割り込みフラグ解除
  PCIRegPointer->CTRL_0 = CTRL0_ICLR;
  pExtension->IntLED = PCIRegPointer->LED;
  pExtension->IntStatus = StsReg;
  PCIRegPointer->LED = 0x0E;

  // Event 発行 必ずしも必要ない
  IoRequestDpc( pDeviceObject, pDeviceObject->CurrentIrp, NULL );

  // 自分の割り込みの時はTRUEを返す
  return( TRUE );
}
```

● 割り込み処理ルーチンの具体例(**リスト 2.11**)

割り込み処理ルーチンの具体的な処理を説明します．DeviceObjectはパラメータContextに渡されるので，適当な変数にキャストして使用します．

1. 割り込み処理ルーチンに制御が渡ってきたら，まず自身(KIT105)の割り込みであるかを確認します．プッシュスイッチの割り込みでないときは，KIT1050が使用しているPLX9054のDMA完了割り込みかを検査します．DMA完了割り込みは必ずしも使用する必要はありませんが，処理のサンプルとして記述してあります．もしDMA完了割り込みもなければ，IRQを共有している他のデバイスからの割り込みなので，FALSEを返して割り込み処理ルーチンを終了します．
2. DMA完了割り込みのときは，PLX9054の完了割り込みをクリアして処理を完了します(TRUEを返す)．今回はサンプルなので，とくに処理は行いません．

3. KIT1050のプッシュスイッチからの割り込みのときは，割り込みフラグをクリアします．アプリケーションに渡すために，割り込みが発生したときのLEDの値と割り込みステータスを保存します．割り込み処理が完了したことを示すために，LEDの点灯を変更します．最後にアプリケーションに割り込みが発生したことを示すためにDPCルーチンを呼び出す登録を行い，処理を完了します(TRUEを返す)．

● DPCルーチン(**リスト2.12**)

他の処理を考慮しなくてもよい専用システムを除き，割り込み処理ルーチンでは時間がかかる処理はマナーとして行わないことになっています．

そこで，割り込み処理ルーチンでは割り込みを検出して最小限の処理を行い，時間のかかる処理はDPCルーチンに処理を依頼します．

今回は，KIT1050のプッシュスイッチが押されたときに，アプリケーションに通知する簡単な方法を解説します[注2.2]．なお，DeviceObjectは，KIT1050DpcRoutine()のパラメータDeferredContextに渡されます．

1. WaitQueueが0ならアプリケーション側でウェイト中のイベントがあることを示しています(①)[注2.3]．
   もし，アプリケーションでウェイト中のイベントがあるときは，リリースするIRP(I/O Request Packet)をExInterlockedRemoveHeadList()によりWaitQueueから取り出します(②)．
2. 新しいIRPを取得したら，登録されているキャンセルルーチンをIoSetCancelRoutine()により無効にします(③)．
3. xxxxCancelSpinLock()は，マルチCPUのときに他のCPUが取り出したIRPへのアクセスを禁止するためです．一つのCPUが前提のときはなくてもかまいませんが，後々のことを考えれば入れておくのが安全です(④)．
4. IoGetCurrentIrpStackLocation()により，リリースするIRPからアプリケーションに渡すパラメータを取り出します(⑤)．
5. 今回は，アプリケーションに対して，割り込みの発生を割り込み発生時のLEDの状態とステータスレジスタの内容を4バイトで通知するのでその処理を行います(⑥)．
6. 最後に，リリースするIRPのPENDING状態を解除して，アプリケーションがウェイト中のイベントを起動させます(⑦)．
7. 以上をリリース要求されているものすべてに対して行います(①)．

〔**リスト2.12**〕**DPCルーチン**(INTSERVI.C/KIT1050DpcRoutine())

```
VOID KIT1050DpcRoutine(
    IN PKDPC Dpc,
    IN PVOID DeferredContext,
    IN PVOID SystemArgument1,
    IN PVOID SystemArgument2
    )
{
  PDEVICE_OBJECT DeviceObject;
  PDEVICE_EXTENSION pExtension;
  PLIST_ENTRY head ;
  KIRQL kCancelSpin ;
  PIRP NewIrp ;
  PIRP Irp;
  PIO_STACK_LOCATION irpStack;
  PULONG  pData;
  LONG  outputBufferLength;
  union  ULONGB  Param;

  DeviceObject = DeferredContext;
  Irp = DeviceObject->CurrentIrp;
  pExtension = DeviceObject->DeviceExtension;

  while( !IsListEmpty(&pExtension->WaitQueue) ) ←①
  {
    IoAcquireCancelSpinLock(&kCancelSpin); ←④
    head = ExInterlockedRemoveHeadList (
        &pExtension->WaitQueue, &pExtension->QueueSpin) ; ←②
    NewIrp = CONTAINING_RECORD (head, IRP, Tail.Overlay.ListEntry) ;

    IoSetCancelRoutine(NewIrp, NULL) ; ←③
    IoReleaseCancelSpinLock (kCancelSpin) ; ←④
    irpStack = IoGetCurrentIrpStackLocation(NewIrp); ←⑤
    outputBufferLength =
      irpStack->Parameters.DeviceIoControl.OutputBufferLength;

    if ( outputBufferLength >= sizeof(LONG) ) {                              ┐
      pData = (PULONG)NewIrp->AssociatedIrp.SystemBuffer;                     │
      Param.dword = 0;                                                        │
      Param.ul.byte_0 = (UCHAR)(pExtension->IntLED & LEDMASK);                ├⑥
      Param.ul.byte_1 = (UCHAR)(pExtension->IntStatus & STS_MASK);            │
      *pData = Param.dword;                                                   │
      NewIrp->IoStatus.Information = sizeof(LONG);                            ┘
    } else
      NewIrp->IoStatus.Information = 0;
    NewIrp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest (NewIrp, IO_NO_INCREMENT); ←⑦
  }
  pExtension->nCount = 0;

  if( Irp ) {                                                  ┐
    // need to fill in this field to get the I/O manager       │
    // to copy the data                                        │
    // back to user address space                              │
    Irp->IoStatus.Information = 0;                             │
    Irp->IoStatus.Status = STATUS_SUCCESS;                     ├←⑧
                                                               │
    IoCompleteRequest(Irp, IO_NO_INCREMENT);                   │
//  IoStartNextPacket(DeviceObject, FALSE);                    │
  }                                                            │
  return;                                                      ┘
}
```

8. 最後に，カレントのIRPを正常終了させて，DPCルーチンを完了します．

まるやま・はるお　ドライバ屋

注2.2：このイベントの登録方法は次回説明する予定．
注2.3：Whileでループの判定に使用しているWaitQueueの意味については次回説明する予定．

第23回 開発環境探訪

C/C++, C#, Javaなどの良いところを取り込んで進化する新たな言語 —— D言語

水野貴明

今回紹介するのは，D言語である．これは，1999年12月に生まれた比較的新しい言語で，CやC++，C#，Javaなどの流れをくんでいる．そして，それらの言語の良いところを取り込み，より使いやすいものとなるべく進化を続けている言語なのである．

インストールと実行方法

D言語のコンパイラは，Digital Mars C++(かつてのSymantec C++)を公開しているDigital Mars社より，Windows版とLinux版が無償で公開されている(ちなみにD言語開発者のWalter Bright氏は，Digital Mars C++の開発者でもある)．まずは，それらのインストールとプログラムのビルド方法を紹介する．

● Windows版の場合

コンパイラとリンカをダウンロードする．コンパイラ(D compiler)とリンカ(linker and utilities)は別々に配布されているので，その両方をダウンロードし，それらのZIPファイルを展開する．コンパイラを展開すると，dmdとdmという二つのフォルダが展開されるが，そのうちdmdフォルダだけを，たとえば「C:¥dmd」のような場所にコピーする．続いてリンカを展開したdmディレクトリを同じく「C:¥dm」のような場所にコピーする．そして「C:¥dmd¥bin」，「C:¥dm¥bin」の2か所にパスを通せば，インストールは完了である．

● Linux版の場合

Linux版は，リンカはGCCのものを利用するため，コンパイラだけをダウンロードする(コンパイラはWindowsとLinux両方が同じパッケージに入っている)．ダウンロードしたZIPファイルを適当な場所(ユーザーのホームディレクトリなど)に展開し，dmd/libにあるlibphobos.aを/usr/libにコピーする．続いて，dmd/libにあるdmd.confをエディタで開き，以下の行をphobosのモジュールが置かれたディレクトリに書き換える．

```
DFLAGS=-I/home/mizuno/dmd/src/phobos
```

その後，dmd.confファイルを/etcにコピーする．そして，同じくdmd/libにあるdmd，obj2asm，dumpobjに実行権限をつけて，このディレクトリにパスを通せば準備は完了である．

*　　　　　　*

D言語のプログラムは「.d」という拡張子をもつ．プログラムのビルドは非常に簡単で，次のようにdmdというコマンドを使えばよい．

```
dmd hello.d
```

するとプログラムのコンパイルとリンクが行われ，実行ファイルが生成される．複数のファイルをまとめてビルドする際は，ただ単にファイル名を並べていけばよい．作成された実行ファイルは完全なネイティブアプリケーションで，バーチャルマシンやランタイムライブラリなどはとくに必要ない．実行ファイルのサイズは，小さなプログラムであればWindows版で60Kバイト程度，Linux版で100Kバイト程度になる．

DATA

名称：D Programming Language
作者：Walter Bright氏
Webサイト：
http://www.digitalmars.com/d/alpha.html
現在のバージョン　：0.69(2003年8月15日現在)
ダウンロードサイズ：1.86Mバイト(コンパイラ)
実行：コンパイラ
OS　：Windows/Linux

基本的なD言語の言語仕様

まずは，基本的なD言語のプログラムを**リスト1**に示す．これは，指定したファイル名のファイルを標準出力に書き出す，簡易的なファイルビューワである．

D言語のプログラミングのスタイルは，C/C++やJavaなどに非常によく似ている．それらの言語を知っていれば，**リスト1**のような短いプログラムならば，読み下すのにそれほど問題ないのではないだろうか．

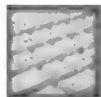
プログラムの流れを簡単に追ってみよう．まず，プログラム

〔リスト1〕基本的なD言語のプログラム

```
import stream;

int main (char[][] args)
{
    File f;
    char[1] input;

    if( args.length < 2 ){
        stdout.writeLine("ファイル名を指定してください.¥n");
            return 0;
    }
    f = new File();
    f.open(args[1],FileMode.In);
    for (ulong i = 0; i < f.size(); i++){
        f.readBlock(input,1);
        stdout.write( input[0] );
    }
    f.close();
    return 0;
}
```

のエントリポイントはmainであり，パラメータが配列変数に入れられて呼び出される．各行の最後にはセミコロン「;」がつく．大文字と小文字は区別される．「import」はモジュールと呼ばれるライブラリを読み出すしくみで，ここでは「stream」という入出力関係の標準モジュールを呼び出している．

D言語はオブジェクト指向言語である．ここではFileというファイルの入出力を行うクラス（streamモジュールで定義されている）を定義している．インスタンスを新しく生成するには「new」を利用し，メソッドの呼び出しは，オブジェクト名とメソッド名を，f.openのようにピリオドでつなぐ．

変数は「char[1]　input;」のように，「変数型 変数名」といった形で定義される．if文，for文などを利用した制御構造も，C/C++とほぼ同じである．

変数の扱い方

さて，D言語は非常に多くの機能をもつ言語である．そのすべてを紹介することは，限られた誌面では難しいが，ここから先はいくつかの機能をピックアップして紹介していく．

まずはD言語における変数の扱い方を紹介する．D言語における変数は，「変数型」の概念をもつ．利用できる変数型を**表1**に示す．変数を定義する際には，C/C++と同様に，「変数型 変数名」と指定する．次のようにアスタリスク（*）を利用してポインタ変数を作ることもできる．

```
int dat;
int dat2 = 100;
int* pointer;
```

D言語では，変数の定義時に初期値をセットしなかった場合でも，値は不定にならない．各変数型には初期値が定義されており，変数は定義された時点ではその値として初期値がセットされる．標準の変数型はすべて初期値は0になっている．

D言語では，すべての変数で「プロパティ」を利用できる．プロパティは，変数名の後にピリオドをつけて指定する．たとえばinitはその変数型の初期値を取得するプロパティなので，次のように「dat.init」と指定すれば，datの初期値が取得できる．初期値としては，変数の定義の際にセットした値か，セットされていない場合は，その変数型の初期値が取得できる．また，「int.init」のように変数型に直接プロパティをつけても，その変数型自体のプロパティを取得できる．

```
int dat;
int dat2 = 100;
printf("%d¥n",dat.init);  // 0が表示される
printf("%d¥n",dat2.init); // 100が表示される
printf("%d¥n",int.init);  // 0が表示される
```

整数型の場合は，**表2**のようなプロパティが用意されている．

複素数を扱う場合は，虚数単位をiとして記述することができる．また，複素数を扱う変数には，re（実数部分），im（虚数部分）というプロパティを利用して，それぞれの値にアクセスすることが可能になっている．

```
cfloat v1, v2, r;
v1 = 10 + 5i;
```

〔表1〕D言語で利用できる変数型

型	説明
void	型なし
bit	1ビット
byte	符号あり8ビット
ubyte	符号なし8ビット
short	符号あり16ビット
ushort	符号なし16ビット
int	符号あり32ビット
uint	符号なし32ビット
long	符号あり64ビット
ulong	符号なし64ビット
cent	符号あり128ビット（現在は利用不可）
ucent	符号なし128ビット（現在は利用不可）
float	32ビット浮動小数
double	64ビット浮動小数
real	ハードウェアが扱える最大サイズの浮動小数（Intel CPUでは80ビット）
ireal	虚数を扱うことができるreal型
ifloat	虚数を扱うことができるfloat型
idouble	虚数を扱うことができるdouble型
creal	二つのreal型を用いた複素数
cfloat	二つのfloat型を用いた複素数
cdouble	二つのdouble型を用いた複素数
char	符号なし8ビットASCII文字
wchar	符号なしwide char型（Win32システムは16ビット，Linuxでは32ビット）

〔表2〕整数型の変数で利用できるプロパティ

プロパティ	説明
.init	初期値
.sizebyte	サイズ（バイト数）
.max	最大値
.min	最小値
.sign	符号

〔表3〕配列で利用できるプロパティ

size	配列のデータサイズ(要素数×各要素のバイト数)を返す
length	配列の要素数．動的配列では要素数を設定することもできる
dup	新しい動的配列を作成し，そこにデータをコピーする
reverse	配列の値を逆順に並べなおす
sort	配列の値を昇順にソートする

```
v2 = -2 + 3i;
r = v1 + v2;
printf( "%f + %fi¥n", r.re, r.im );
            // 8.000000 + 8.000000i と表示される
```

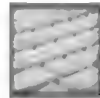
さらにD言語では，typedefを使うことで，新しい変数型を定義することができる．

```
typedef int flag = -1;
```

上記の場合，「flag」というint型と同じタイプで，初期値が−1である変数型を定義している．typedefを使って定義された変数型は，単なる型の別名ではなく，元となった変数型とはまったく異なる変数型として認識される．デバッガや関数のオーバロード(後述)の際などにも，まったく異なる変数型として認識される．

typedefとは別に，変数型の単なる「別名」を定義するaliasも用意されている．

```
alias int flag;
```

typedefが完全に別の型として扱われるのに対し，こちらの場合は，コンパイラはintとflagという二つの変数型はまったく同じ型として扱う．つまり同じ型に単に別の名前をつけただけとみなされるのである．

配列の定義

D言語では，静的配列と動的配列の両方がサポートされている．配列を定義する場合は，次のように宣言をする．

```
int[10] a;  // 要素数10の静的配列
int[] b;    // 動的配列
```

要素数を指定すれば静的配列に，指定しなければ動的配列となる．配列の要素数はlengthというプロパティで取得できる．配列には，ほかにも**表3**のようなプロパティが用意されている．配列のソートなども，プロパティ一つで簡単にできるのが特徴的といえるだろう．

D言語の配列は，「スライシング」と呼ばれる配列のうちの一部を範囲指定する手法が利用できる．これは[1..3]のように要素を範囲として指定するもので，たとえば，次の例は変数bに変数aの1〜3の要素だけをもつ要素数3の配列に設定する．

```
int[6] a;
int[] b;
b = a[1..3];
```

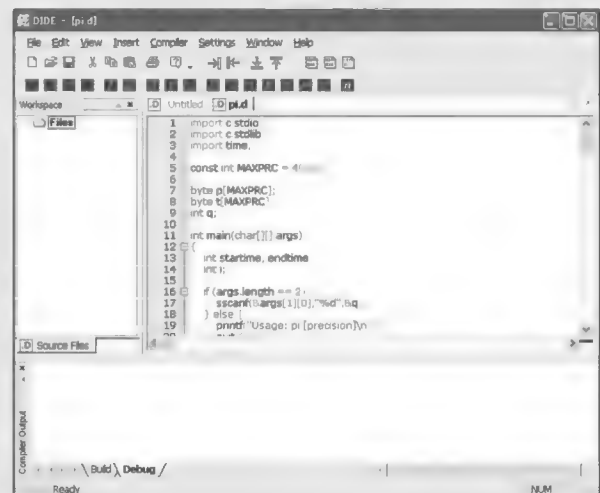
Column 統合開発環境DIDE

Digital Marsで公開されているD言語の開発ツールは，すべてコマンドラインで実行するものだが，Charles Sanders氏によって，DIDE("D" IDE)というWindows用の統合開発環境が公開されている．DIDEは，GUIのインストーラが用意されており，簡単にインストールできる．また，dmd(D言語のコンパイラ)とdigライブラリが同時にインストールできるインストーラも公開されており，こちらを利用すれば，インストール作業をすべてGUIで行うこともできる．ただし，現在コンパイラのバージョンアップの速度が速いため，同梱されるコンパイラが，Digital Marsで配布されているものよりも若干古い場合もあり，最新版のコンパイラを試したい場合は，DIDEとコンパイラは別々にインストールを行ったほうがよいだろう．

DIDEの開発環境は，複数のファイルを同時に開けるタブ形式になっているほか，キーワードのハイライトや，よく利用するフレーズを簡単に挿入する機能，プロジェクトファイルを利用して複数のファイルを管理する機能なども備えており，かなり使いやすい開発環境となっている．WindowsヘルプとしてPhobosのリファレンスが呼び出せるのもなかなか便利である．今回の検証においても，プログラミング作成はすべてこのDIDEで行っている(**図A**)．

「Settings→Editor Settings→Change Font」からフォントを日本語フォントに変更すれば，日本語の表示や入力も可能である(ただし削除の際，DELキーを2回押さないと，全角文字の削除ができない．これは，海外で作成されたエディタにはよくある挙動である)．

● Atari-Soldiers(DIDEの配布元)のサイト
http://www.atari-soldiers.com/

〔図A〕DIDEでのプログラム編集中の画面

〔図1〕配列を別の配列に代入すると，同じメモリを共有する

なおこの場合，bのために新しい領域が用意されて値がコピーされるのではなく，**図1**のように，配列Aと配列Bがメモリを共有することになる．

データをコピーするには，dupプロパティを利用する．

```
b = a[1..3].dup;
```

また，左辺にスライシングされた配列を指定することもできる．左辺にスライシングを利用した場合も，右辺のデータが左辺にコピーされる．ただし，右辺と左辺の要素数が一致していない場合はエラーが発生してしまう．

```
b.length = 2;    // まずbのサイズを決定(0～2)
b[] = a[1..3];   // それからコピー
```

さらにD言語では，連想配列(要素を文字列で特定できる配列)を利用することもできる．

```
int[char[]] aa;
aa["hello"] = 5;
printf( "%d¥n", aa["hello"] );
```

関数

続いては，関数である．まずは，簡単な関数のサンプルを次に示す．

```
void MyFunc( int dat1, out int dat2 ){
    dat2 = dat2 + dat 1;
}
```

D言語におけるもっともシンプルな関数は，C/C++のそれとよく似ている．しかし関数に関しても，D言語ならではのユニークな特徴もいくつもある．

まず，関数のパラメータにある「out」だが，これは変数が参照渡しで関数に渡されることを意味する．D言語にはパラメータの渡し方として「in」，「out」，「inout」がある．inはいわゆる値渡し，outは参照渡しを意味し，省略すればinであると解釈される．ユニークなのは「inout」で，これは参照渡しなのだが，関数が呼び出された際に初期値で初期化される．したがって，そのパラメータに変数を渡した場合，その変数に呼び出し前にどんな値が入っていたとしても，関数が呼び出された時点で初期値になる，というものだ．ところが，現在のコンパイラでは，inoutにはバグがあり，値が初期化されず，outとまったく同等の機能となってしまっている．頻繁に使うものではないが，なかなか面白い機能なので，ぜひ早く利用できるようにしてもらいたい．

〔リスト2〕ネストした関数のサンプル

```
import stream;import c.stdio;

int main(char [] [] args) {
    int b;
    b = MyFunc1(6);
    printf("%d¥n", b); return 1;
}
int MyFunc1(int a)
{
    int x=12;
    int MyFunc2(int b)
    {
         int MyFunc3(int c){
              return x+c;
         }
         return b + MyFunc3(5);
    }
    return MyFunc2(a);
}
```

〔リスト3〕関数のオーバロード

```
import stream;import c.stdio;

typedef int myint;

int MyFunc(int a){ return 10; }
int MyFunc(myint a){   return 20; }

int main(char [] [] args) {
    int a;
    myint b;
    printf("%d¥n", MyFunc(a)); // 10が表示される
    printf("%d¥n", MyFunc(b)); // 20が表示される
    return 1;
}
```

さて，D言語では関数の中に関数をネストして記述することができる(**リスト2**)．このサンプルでは，MyFunc1という関数の中に，MyFunc2という関数が定義され，さらにその中でMyFunc3という関数が定義されている．このように関数がネストされた場合，ある関数Aの内部に書かれた関数Bは，その関数A内でしか呼び出すことができない．つまり**リスト2**では，MyFunc2はMyFunc1の中でしか，MyFunc3はMyFunc2の中でしか呼び出すことができない．ただし，ネストされた関数内では，その関数を内包する関数のローカル変数にアクセスすることができる．**リスト2**では，MyFunc3の中で，MyFunc1のローカル変数であるxにアクセスしている．

また，関数はオーバロードが可能である．**リスト3**のように，異なるパラメータ，戻り値をもつ同じ名前の関数を定義しておくと，関数呼び出し時に，与えられたパラメータや戻り値の変数型によって，適切な変数型のパラメータ/戻り値をもつ関数が選択される．変数の項でも述べたが，この際typedefで定義した変数型は，元の変数型と別物として扱われるが，aliasで定義した別名は，同じ変数型として扱われる．

モジュール

D言語ではimport文を利用して，別のファイルとして提供されたさまざまな機能を利用できる．これらのファイルは

「モジュール」と呼ばれ，一つのモジュールは一つのファイルで構成されている．たとえば，次のように宣言することで，そのファイルに含まれる機能が利用可能になる．

```
import stream;
```

D言語にはPhobos(フォボス)注1と呼ばれる標準ライブラリがついており，「C:¥dmd¥src¥phobos」というパスにモジュールファイルが入っている．たとえば上記の「stream」というモジュールは，もっともよく利用するモジュールの一つで，入出力関係の処理に関する機能を提供するものだが，実体は「C:¥dmd¥src¥phobos¥stream.d」というファイルである．

また，もう一つよく利用されるモジュールは「c.stdio」である．こちらは，ANSI Cのstdioライブラリと互換の機能を提供するものである．

```
import c.stdio;
int main (char[][] args)
{
    printf ("Hello World!");
    return 0;
}
```

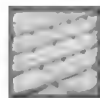
このモジュールの実体はC:¥dmd¥src¥phobos¥c¥studio.dになる．「c.stdio」の「c」はディレクトリの名前を表しており，パッケージ名と呼ばれる．cというパッケージには，このほかに「c.stdlib」というモジュールが用意されている．こちらは，その名のとおりANSI Cのstdlibライブラリの機能を提供するモジュールとなっている．

〔リスト4〕クラスのサンプル

```
import c.stdio;

class human{
    this(){
        printf("Hello!¥n" );
    }
    void talk(){
        printf("I am human.¥n" );
    }
    ~this(){
        printf("Bye-bye!¥n" );
    }
}

class Tommy : human { // humanを継承してTommyクラスを作る.
    this(){
        printf("Tommy:" );
        super();
    }
    void talk(){
        printf("I am Tommy.¥n" );
    }
}

int main(char [] [] args) {
    Tommy t;
    t = new Tommy; // "Tommy:Hello!"と表示される
    t.talk();      // "I am Tommy."と表示される
    return 0;      // インスタンスが破棄され，"Bye-bye!"と表示される
}
```

モジュール内で定義されている変数や関数，クラスなどは，importすればそのまま利用できる．たとえば上記のc.stdioの例では，printfという関数を利用している．しかし，複数のモジュールを利用している場合で，両方に同じ名前の変数や関数が使われている場合は，単に変数・関数名を書いただけでは，どちらのモジュールのものか特定できずに，エラーとなってしまう．たとえば，streamモジュールとfileモジュールにはどちらもreadという関数があるので，両方を利用する場合には，次のようにモジュール名と関数名をピリオドでつないで表現する．

```
bite[] dat = file.read(filename);
```

モジュールはD言語で記述されており，新しいモジュールを自分で作成することももちろんできる．モジュール内で別のモジュールをimportすることも可能である．C/C++のインクルードファイルとは異なり，複数回同じモジュールがimportされても，問題なく処理される．

クラスの利用

すでに述べたように，D言語はオブジェクト指向の概念が取り入れられており，クラスを扱うことができる．クラスを利用した基本的なプログラムを**リスト4**に示す．

クラスのコンストラクタ，デストラクタはそれぞれthis，~thisという名前で定義する．もちろんクラスの継承も可能だが，多重継承は許可されていない．インスタンスはnewを使って生成し，メンバを呼び出す際にはインスタンス名とメンバ名をピリオドでつなぐ．D言語にはガベージコレクタが実装されているので，参照されなくなった時点でクラスは破棄され，デストラクタが実行される．

D言語のクラスのユニークな点は，this，~this以外にも，いくつか特定の意味をもつ特別なメンバ関数名がある点である．

- クラスアロケータ/クラスデアロケータ

クラスアロケータとクラスデアロケータは，インスタンスへのメモリ割り当てを明示的に行うことを可能にする．クラスアロケータはnewという名前の関数で，パラメータとしてインスタンスの作成に必要なメモリサイズを受け取る．そして，戻り値はポインタであり，この関数の中では，指定されたサイズのメモリを確保し，それを返す必要がある．逆にクラス デアロケータは使い終わったメモリを渡され，それを破棄するために利用される(**リスト5**)．クラスアロケータはコンストラクタよりも前，クラスデアロケータはデストラクタよりも後に呼び出される．

- クラス不変条件

クラス不変条件は，invariantという名前で，コンストラクタ呼び出し直後，デストラクタの呼び出し直前，メンバ関数(public/exportの属性のもののみ)の実行の前後に呼び出さ

注1：フォボスは火星にある二つ衛星のうち大きいほうの名前．D言語を配布している「Digital Mars」(Marsは火星)の名前に由来する．

〔リスト5〕クラスアロケータとクラスデアロケータ

```
class MyClass
{
    new(uint sz){
        void* p;
        // pにszのメモリを割り当てる処理
        return p;
    }
    delete(void* p){
        if (p){
            // メモリを破棄する処理
        }
    }
}
```

〔リスト7〕単体テスト

```
class Sum
{
    int add(int x, int y) { return x + y; }
    unittest
    {
        assert(add(3,4) == 7);
        assert(add(-2,0) == -2);
    }
}

int main(char [] [] args) {
    Sum s;
    s = new Sum;
    return 1;
}
```

〔リスト6〕クラス不変条件

```
class Date
{
    int day;
    int hour;

    this( int in_day, int in_hour ){
        day = in_day;
        hour = in_hour;    }
    invariant
    {
        assert(1 <= day && day <= 31);
        assert(0 <= hour && hour < 24);
    }
}

int main(char [] [] args) {
    Date d;
    d = new Date(10,26); // in_hourが24より大きいので実行時に例外が発生
    return 1;
}
```

〔表4〕D言語で利用できる属性

Linkage()	利用する呼び出し規約を指定する．Linkage(C)ならC言語の呼び出し規約．extern(Windows)ならWindowsの呼び出し規約を利用する．C/C++/PASCAL/D/Windowsの5種類が利用できる
Align()	構造体のメンバのアラインメントを指定．Align(8)なら，8バイト単位で整列される
deprecated	後方互換性のために残されたものであることを指定
private	同じクラスのメンバのみ，もしくは同じモジュールのクラスや関数のみが参照できることを指定
protected	同じクラスかその派生クラスのメンバからのみ参照できることを指定
public	実行ファイル中のすべてのコードからアクセスが可能であることを指定
export	実行ファイル外からも呼び出しが可能であることを指定
static	関数，変数が特定のインスタンスに依存しないことを指定
final	クラスをそれ以上継承できなくする
override	クラスのメンバ関数などで，その関数が継承したクラスの関数をオーバライドしていることを明示する．もし，オーバライドすべき関数が継承もとのクラスになければ，エラーとなる
const	指定した変数が，定数として扱われ，コンパイル時に評価されるようになる
auto	クラスやローカル変数で，スコープを外れた際にガベージコレクタによってリソースが破棄されることを明示する

れる関数である．これは，現在のクラスのメンバ変数などの状態が，ルール違反になっていないかをチェックするために利用される．たとえば，**リスト6**のようにして利用する．assertはその内部の式がTrueにならないと例外が発生する．これはテスト用の関数なので，コンパイル時に-releaseというオプションをつけることで，invariantはコンパイルされなくなる．

● 単体テスト

unittestという関数は，そのクラスの単体テストを行うことができる．これは，そのクラスが単体できちんと動作するのかをチェックするもので，やはりassertで例外を発生させることでチェックを行う（**リスト7**）．プログラムが実行されて初期化が済み，main関数が呼び出される直前に，定義されているすべてのクラスのunittestが呼び出されるようになっている．invariantと同様，テスト用の関数で，コンパイル時に-unittestというオプションをつけることで，単体テストが実行されるようになる．

属性（attribute）

D言語では，変数や関数，クラスに属性をつけることができる．属性は，その変数/関数，クラスのアラインメントを指定したり，後方互換性のために残された「利用すべきでない」ものに印をつけたり，といったそれらの変数/関数/クラスの性質を設定するものである．D言語で利用可能な属性を**表4**に示す．

たとえば**リスト8**では，dateという変数にprivate属性を，Addという関数をdeprecated属性をつけている．こうすることで，変数dateには，このファイル外からはアクセスができなくなり，Addを呼び出していると，コンパイル時にエラーが出るようになる．deprecatedされた物を利用した場合に出るエラーは，コンパイル時に-dオプションをつけることで，表示させないようにすることもできる．

契約（contract）

契約は，式の結果がTrueでなくなった場合に，例外を発生させることができるしくみである．まずは，**リスト9**のサンプルプログラムを見ていただきたい．ここで定義されているadd_oneという関数は，契約を利用した記述方式を採用して

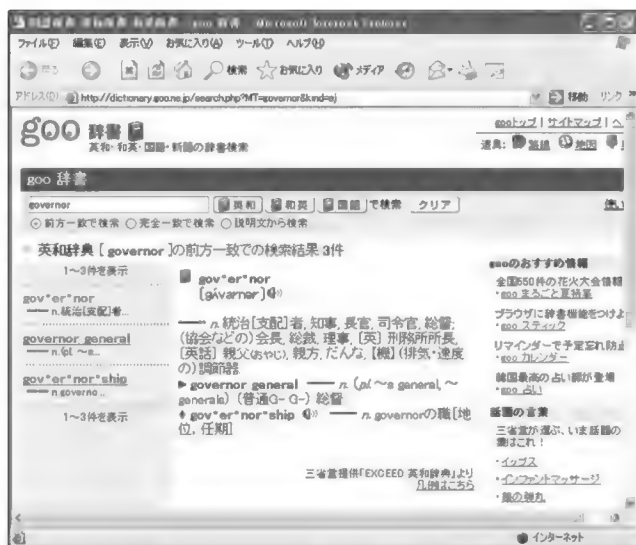
〔リスト8〕属性をつける

```
private int date = 200;

int Add_data( int a, int b ){
    if( a>0 && b>0 ){
        return a+b;
    }else{
        return 0;
    }
}

deprecated {
int Add( int a, int b ){
    return a + b;
}
}

int main(char [] [] args) {
    int s = Add( 1,3);
    return 1;
}
```

〔リスト9〕契約のサンプル

```
import c.stdio;

int add_one(int x)
    in
    {
        assert(x >= 0);
    }
    out (result)
    {
        assert(result <= 10);
    }
    body
    {
        return x + 1;
    }

int main(char [] [] args) {
    int a;
    a = add_one(5);
    printf( "%d¥n", a);return 1;
}
```

〔図2〕
英単語検索ツールの実行画面

〔図3〕ボタンを押すとウェブブラウザが立ち上がり検索結果が表示される

いて，in，out，bodyという三つのパートで構成されている．まずbodyは，関数の処理を記述した関数の本体である．そして，inはこの関数が実行される直前，outは関数が実行された直後に実行され，入力，出力の値などが正しい値であるかをチェックする部分となっている．

正しい値であるかどうかのチェックは，クラス不変条件などと同様にassertを利用する．**リスト9**のサンプルでは，関数が呼び出された際に，渡されたパラメータが0より小さければ例外となり，また計算結果が10を超えた場合にも例外となる．したがって，この関数に渡せる値は0から9までの間の数だけとなるわけである．

GUIプログラミングへの対応

D言語では，標準ではGUIプログラミングには対応していないが，Burton Radons氏により，「dig Library」というGUIライブラリが公開されている(http://www.opend.org/dig/index.html)．digは，最終的にはOS非依存のGUIライブラリとなることをめざしているが，現在はWindowsのみで動作する．

dig Libraryのパッケージを展開すると，中に「go.bat」というバッチファイルが含まれている．これを実行すれば，dig Libraryが適切にインストールされる．ただし，「C:¥dmd」以外の場所にコンパイラをインストールしてある場合，同じディレクトリにあるmakefileの中の先頭の「BASEDIR = c:」を適切な場所に変更する必要がある．

ここで，このdig Libraryを使って，「英単語検索ツール」を作成してみた．これは起動すると，**図2**のようなテキストボックスとボタンが配置されたウィンドウが表示される．テキストボックスに適当な英単語を入力してボタンを押すと，Webブラウザが立ち上がり，インターネット上の英単語検索サービス(今回はGooのサービスを利用している)での，指定した単語の検索結果が表示される(**図3**)というものだ．

このプログラムのソースコードを**リスト10**に示す．dig Libraryは，ウィンドウやボタン，テキストボックスといったコントロールがすべてクラスとして定義されている．通常はウィンドウを扱うFrameクラスを継承したウィンドウクラスを作成して，その中に各パーツを配置し，そのウィンドウを表示するというスタイルを用いる．**リスト10**では，MainWindowというクラスを定義している．コンストラクタ内では，ウィンドウのタイトルを設定し，テキストボックスとボタンのインスタンスを作成する．画面上のパーツの作成時には，そのパーツ

〔リスト10〕英単語検索ツールのプログラムリスト

```
import net.BurtonRadons.dig.main;

extern(C) int ShellExecuteA(int, char*, char*, int, int, int);

class MainWindow : Frame
{
    Button search;
    EditText word;

    this () {
         caption ("EtoJ Search");     // ウインドウのタイトル
         word = new EditText(this);   // テキストボックスの作成
         word.grid(0,0);
         word.width(150);
         search = new Button(this);   // ボタンの作成
         search.grid(0,1);
         search.sticky("<>");
         search.caption("Search");
         search.onClick.add(&openbrowser);
    }
    void openbrowser(){
         Control.OS sys;
         int err;
         sys = new Control.OS;
         char [][] arg;
         arg.length = 1;
         arg[0] = "http://dictionary.goo.ne.jp/
search.php?MT="~word.text()~"&kind=ej" ;
         err = sys.system("explorer", arg );    }
}

int main( char[][] arg )
{
    (new MainWindow).showModal ();
    return 0;
}
```

を貼り付けるウィンドウのインスタンスをコンストラクタのパラメータとして指定する．

ウィンドウ上の各パーツの配置は，ちょうどJavaにおけるGridLayoutのように，画面上を格子状に区切り，grid(0, 0)やgrid(1, 2)といったそのマス目の位置で指定することになる．今回の場合はテキストボックスを(0, 0)に，ボタンを(0, 1)に指定することで，上下に並べて配置している．テキストボックスはwidthというプロパティで幅を150ピクセルに指定しているが，ボタンのほうはstickyというプロパティを使ってサイズを指定している．これはグリッド枠内の右と左，どちら側よりにパーツを表示するかを指定するもので，「<>」を指定すると，グリッド枠一杯にパーツが配置される．テキストボックスを150ピクセルに指定したことで，真下にあるボタンの置かれたグリッド枠も150ピクセルになっているので，150ピクセルいっぱいにボタンが描画されることになる．

ボタンが押されたときの処理は，openbrowserという関数で定義されている．この関数をonClickというイベントに結びつける(addメソッド)ことで，ボタンが押されたときにこの関数が呼び出されるようになる．

英単語の検索は，実際には英単語が検索されたときのURLをプログラム内で生成し，それをパラメータとしてExplorerを起動するコマンドを実行しているだけである．するとWebブラウザ(Internet Explorer)が自動的に起動し，そのURLが表示されることになる．Explorerの起動には，やはりdig Libraryで用意されているControl.OSというクラスを利用する．このクラスのメンバ関数であるsystemを利用すると，コマンドを実行できる．

最後に，main関数内で，定義したMainWindowクラスのインスタンスを作成し，showModalで表示している．

ちなみに，dig Libraryを使ったプログラムをコンパイルするには，digcというコマンドを利用する．これは，dig Libraryと同時にインストールされるコマンドで，dig Libraryを利用するのに必要なライブラリの指定などを自動で行ったうえで，dmdを呼び出してくれるコマンドである．自分ですべてのライブラリを指定したうえでdmdを呼び出すことも可能だが，digcを利用したほうが便利である．

おわりに

D言語の特徴を駆け足で見てきたが，いかがだっただろうか．これまで見てきたように，D言語はC/C++，Java，C#といった既存の言語を研究し，その良いところをどんどん取り込んだ言語である．作者のWalter Bright氏は，Digital Mars C++(以前はSymantec C++，さらに前はZortech C++と呼ばれていた)をずっと開発してきた人物で，それらの開発における経験や反省が，D言語に反映されているようだ．解説した以外にもさまざまな機能が実装されているので，興味のある方はぜひ配布元のマニュアル[注2]を参照してほしい．K.INABA氏によって日本語訳[注3]も公開されている．

現在はまだ発展途上で，仕様も刻々と変化していたり，作成したプログラムが仕様どおりに動かない部分もあるなど，まだまだ未発達な部分もあるが，現状でも非常に豊富な機能をもち，十分実用に耐える言語であるといえる．

ちなみに「D言語」という名前は，C言語の後継という意味が込められている．もともとは「Mars」という名前だったが，Walter Bright氏の友人がD言語と呼んでいたため，結局この名前になったとFAQ[注4]に記されている．D言語，というとちょっと大それた名前であるようにも感じるが，もしかしたら非常にポピュラーな言語になっていくかもしれない．そんな可能性を非常に感じさせてくれる，今後がとても楽しみな言語である．

みずの・たかあき

注2：http://www.digitalmars.com/d/
注3：http://www.kmonos.net/alang/etc/d/html/index.html
注4：http://www.digitalmars.com/d/faq.html

● 頭は良いが……

日本人は，人を大きく二つに分類して考えるクセが付いてしまっているようだ．その二つとは，「理系」と「文系」である．たいていの高校では高二か高三になると，理系と文系に分かれる．これは，大学受験のため，学習する科目を絞るのが目的だ．このとき，得意科目や志望大学とあわせて選択に悩む人は多いが，この選択がずっと影のようにまとわりつくことに気がつくのは，ずっと後になってからのことかもしれない．

一般的には，数学や物理が苦手な人が文系を選ぶという傾向が強いように思う．そこから，文系より理系のほうが頭が良いという一種の社会通念も生まれる．ベネッセ研究所が高校生に対して行った調査では，理系のイメージは「頭が良い」というものが68.6％もあった．これに対し，文系が頭が良いと思うものは24.6％しかない．注目すべきは，文系の人間も，相方の理系のほうが頭が良いと考えているということだ．

そんな理系は，日本の社会を牛耳っていてもよいはずだ．しかし，実際はそうではない．憲法で定められた三権の主要な位置は文系に握られ，経済界でも安定企業の経営者や管理職は圧倒的に文系出身が多い．理系でもっとも優秀な人材が集まるといわれる医者は，多くの人々を支える裏方（?!）の人材にすぎない．

毎日新聞の「理系白書」によれば，日本の技術職の初任給は，事務職・営業職の1.11倍，研究職は1.18倍と，若干賃金が高い．これは，高校時代から培われた「理系のほうが頭が良い」という得体の知れない優越感に裏付けられたものに違いない．しかし，生涯賃金で計算すると逆転している．理系の生涯年金は，文系のそれの70～80％にすぎない．

金融系の給料が高かった（今でも十分高い!?）ことが，文系の生涯賃金を押し上げているという見方もある．しかし，そんな話で単純に結論付けることはできない．なぜ金融系の給料が高いのか，という疑問に答えていないからだ．

● 労働集約と頭脳集約の違いは決定的

筆者は以前，ソフト開発の会社に勤めていたが，技術者の地位は高かった．なかなか売り上げも上がらず「なんでこれが売れないんだ！」などと責任を追及し，営業責任者の首をすげ替えることもしばしばだった．

その後，筆者は商社に転職し，賃金の高さには大いに満足した．しかし，予想外だったのは，商社における技術者の役目は，「営業サポート」という裏方だったということであった．

商社では，自社で主体的に開発を行っていないからだろう．オリジナル製品を企画して開発すればいいのに，筆者はそう思った．しかし，若い営業マンに一蹴されてしまった．彼にいわせると，「開発の仕事は，ようするに労働集約型事業だ．一方で，営業の仕事は販売戦略を考え，いかにして利益を導くかという頭脳集約型だ」．

実際に，商社の営業マンに支払われている賃金と，以前いたソフト開発の会社の賃金との差をよく知っている筆者は，反論できず，大きなカルチャーショックを受けた記憶がある．

銀行や生保は，他人の金や商品を集め，それを転がして利ざやをかせぐ．利潤は薄いかもしれないが，自社で負担するリスクが少ない分，失敗も少ない．「他人の資源を利用して，大して投資をしなくても安定してそこそこ儲かる」のだ．このような話は，金融系だけにとどまらない．

理系が毎日働く働きバチに徹している間，文系は自ら手を下さずして甘い汁を吸うことを必死に考える……．これは，終身雇用で安定性を好み，他の真似をして世界一になったともいわれた日本には，理想形なのか．そして，理系の社会的地位を落としている理由は，これなのかもしれない．

振り返れば，理系は高校の段階から頭が良いといわれ，人と付き合うことを拒絶し，数学と理科の勉強に没頭する．じつは，一定のパターンの計算を正確にこなすことばかりを勉強していると気付かずにだ．一方で，経営学，政治学といったことをからきし軽視して，社会で生き抜く術を学び損じてしまう．これが，もしかしたら人生でいちばん勉強する時期だというのだから，始末が悪かった．

その後，大学を出て技術者となり，研究・開発職に没頭しても，政治力や調整力が不足しがちの技術者は，機器が不足してもお金を自分で手当てすることすらできない自分に気がつく．

● 海外の「理系」は？

はっきり理系，文系と分けて考えるのは，じつは日本独特のものらしい．たとえばアメリカでは，高校の成績や，民間のテスト機関が実施するSAT（Scholastic Assessment Test），ACT（American College Test）など，高校在学中の試験で大学が決

まることが多いようだ．SATは，英語と計数両面の論理思考試験と科目試験があり，ACTは，英語，数学，読解，科学推論の4科目，つまり理系も文系もどちらの力も要求され，「論理的思考力」が試される点が特徴だろう．

大学卒業後も，技術・研究職は静かな環境で頭脳労働を行う．そういう彼らの仕事は高く評価されており，個室を与えられることが多い．ベンチャー指向は「オリジナリティの高い高利益率のビジネス」を可能にし，彼らの高賃金を生み出す．

既出の毎日新聞「理系白書」によれば，アメリカでは，事務職の平均賃金を1とすると，技術職は1.65倍，研究職はなんと2.13倍と給料が高いのが一般的で，事務・営業職より技術・研究職のほうが，社会的地位は高いのが普通だという．

そういえば，マイクロソフトのビル・ゲイツ会長，サンマイクロシステムズのビル・ジョイ会長，アップルのスティーブ・ジョブス会長，インテルのアンディ・グローブ会長……著名なベンチャー成功企業の経営者はみな，技術系の出身だったりする．

筆者は仕事で，多くのアメリカ人の技術者に会ったことがある．比較的フレンドリーで，アグレッシブな人が多かった．日本のように，お客さんの前でおとなしく，営業さんから「彼は技術者ですから……」などと弁明されるようなことはなかった気がする．

● 世界に通用する技術者をめざそう！

頭が良く器用な人は，理系に多かった気がする．そのため専門家として作業に没頭，人と話すことも少なく周りが見えなくなってしまう．器用貧乏とはよくいったものだ．文系の人たちは，自分ができないことを自覚している分，理系の人をうまく使ってお金をもうけることを考えるのだ．ここまでいうと，負け惜しみに聞こえるかもしれないが．

先ほどのベネッセの調査では，理系は社交性がないというイメージが61.3％もあったが，文系には33.9％しかなかった．今度は，理系には社交性がないことを，理系，文系ともに認めているのだ．もともと日本人は社交性がないといわれるが，この調査から，とくに理系で強く裏目に出てしまうようだ．これでは，年をとっても何の人脈もできず，路頭にさまようことになるかもしれない．

そう考えると，技術者は二つの対策をとる必要がある．一つは，あきらめずに政治学・法律学，マーケティング理論，歴史など，文系の勉強を継続的に行うことである．政治学やマーケティング理論は，仕事をしている際の判断に役立つ．歴史からは人生を生きぬくための多くの経験を得ることができる．勉強する時間のとれない人は，本を読むだけでも充分だろう．こういった文系の教養を身に付けることを常に考えることで，自らの価値を高めることができるはずだ．

もう一つは，社交性を磨くことだ．打ち合わせなどで見ていると，技術者はあまり笑わないことが多い．もっと積極的に笑い，ジョークでも飛ばすべきだ．場所にもよるが，まずはエレベータで乗り合わせた人，バス停で待っているときなどに人に話しかけてみたりと努力するとよいかもしれない．宴会が苦手な人は積極的に参加し，自分を磨く絶好の機会と思うことだ．海外の技術者は意外と社交的で，（日本でいう）文系的要素を多くもっている．これも，世界に通用する技術者になれる条件の一つだと思う．

あさひ・しょうすけ　テクニカルライター
イラスト　森　祐子

Engineering Life in

ユーザーインターフェースのスペシャリスト(第一部)

▌今回のゲストのプロフィール
ブラッド・ホックバーグ(Brad Hochberg):ニューヨーク州ロングアイランド出身.スタンフォード大学にて情報工学科を卒業後,オラクルを経てアップルコンピュータに入社.アップルでは,Copland など MacOS の主要プロジェクトにエンジニアとして参加.その後ユーザーインターフェースのスペシャリストとして VTEL のスタートアップ,OnScreen24 に行く.現在は,パーソナルビデオレコーダで著名な TiVo で,ユーザーインターフェースデザイナとして活躍中.趣味はミュージカルや演劇の鑑賞,料理.最近はサイクリングとヨガに熱中している.

☆ 典型的なパソコン少年

トニー さて今回のテーマは,ユーザーインターフェースというユーザーからもっとも見えやすいところにあるうえに,シリコンバレーでは稀な家電系の会社に勤めているということで,いろいろと面白い話を期待しています.ブラッドは,子供の頃からけっこうパソコンに馴染んでいたほうでしょう?

ブラッド そうですね,小学校の頃からパソコンで遊んでいました.家族が AppleII plus を買ってくれて,それでいろいろプログラミングに馴染んでいったのです.

その頃は新聞配達をやっていたのですが,VisiCalc で顧客データベースを作りました.配達を担当した家々の集金もやるのですが,休暇や出張で長期間不在にする家庭や,帰宅が不規則な人などいろいろいるわけです.それで集金日は全体の三分の一ぐらいしか達成できず,時間の無駄と感じていました.それで顧客データベースを作って,顧客の取っている新聞の種類――たとえば週末だけの人とかもあるし,平日だけという人もいます――に応じて,それぞれの不在の日は配達をしないよう,配達の日程調整に利用しました.また,毎月決まった日に請求書をプリントアウトしてそれぞれの家庭にもって行きました.数日後の集金日にあわせてね.支払いは小切手で私の自宅に送るなり,集金日に払ってもらうかはお客さんまかせにしました.

トニー なるほどね.新聞配達といえばアメリカの小さな町では小学生や中学生のアルバイトの代表みたいなものですが,無駄をなくすためにのプログラミングですか.でも,少しませた小学生ですよね.自分の請求書・明細を作るとは…….結果はどうでした?

ブラッド 私の担当地域で 2 家庭だけこの私の請求書に難色を見せて,新聞社に文句を言ってきた人がいましたが,ほかの方々には好評だったようです.まあ,個人的にいうと以前のように一軒一軒回るとそれなりに私の顔を見るのでチップが出たのですが,請求書になるとチップが少し減ったのが問題なくらいです.でも何度も集金に出なくて済んでよかったですね.

トニー その後は?

ブラッド 中学・高校時代は,アップルコンピュータのディーラーで店員として働いていたので,常に新製品をいじることができて楽しかったですね.レーザプリンタとかが出て DTP がはやり始めた頃は,まだまだハードウェアが高価だったのでお店でいろいろと試したりでき,本当に楽しかったです.

☆ 環境都市学を専攻

トニー それでスタンフォード大学に行かれたときは,もちろんコンピュータ・情報工学の専攻だったのですか?

ブラッド いいえ,環境都市工学科(Urban Studies)の専攻でした.そしていずれは,建築家になる予定でした.

トニー それは意外ですね! 典型的なパソコン少年のような気がしたのですが?

ブラッド コンピュータは私にとって,ホビーや趣味のはんちゅうのものでした.それであまり職業としてやりたいと思えなかったのです.また,実際の情報工学科のベースとなる応用数学の世界などに,あまり興味がわかなかったのです.

トニー 環境都市学ではどんなことをするのですか?

ブラッド 都市開発での建築物とその環境,そしてそれらを利用する人々に関して解析をして新しい提案をしたりします.美術のクラスとかで美的感覚を磨いたり,建築学系のクラスを受けたりもします.授業は,小さなクラスで意見をぶつけ合うディスカッションが多く,とても楽しかったです.

具体的な内容は,たとえばサンフランシスコのダウンタウンに長距離バス,地下鉄と市バスが集結する Trans Bay Terminal というたいへん古びたビルがあります.いずれはシリコンバレーとサンフランシスコを結ぶ列車である CalTrain の終点になり,公共交通網の主要拠点となることが期待されています.ここに行って既存の建物や,まわりの建物,環境そして利用者を観察します.それで新しいビルの企画を期末プロジェクトとして提出するのです.私の提案したプランでは,まわりのビルに配慮して光を通すガラス立てのビルを作り,下位にはバス停と駅を作り上位にホテルや店舗スペースを作る案を提出しました.10 年前の話ですが,まだ何も建設されていないのが皮肉ですよね(笑).

トニー う～ん,州政府やサンフランシスコ市は,予算がまったくないので難しいですよね.

☆ 情報工学科に専念――ユーザーインターフェースとの出会い

トニー 情報工学科に専念したのはどういうきっかけですか?

ブラッド いずれは,建築家になるために専攻したのですが,一人前の建築家になるのにどれぐらいたいへんか,少しずつわかってきたのです.建築家は競争率が激しいし,ライセンスを取るための受験が必要だし,またかりにそこまで行けてもどこかの有名な先生の下でかなりの下働きが必要で,やっと自分の設計ができるまでとても時間がかかると聞きました.それで,そこまでして建築家になるのもどうか?と疑問に思いました.また,このまま環境都市学で卒業すると,市役所とか役所系の

Silicon Valley

H. Tony Chin

対談編

仕事になるので，それもどうかと感じていました．それで，好きなコンピュータの世界に行こうと思ったのです．3年生のときでした．一応両方で学位をとるつもりで両立させようとしたのです．しかし，それぞれの単位を取るためのクラスを両立するのは無理とわかったので，4年生になる頃は情報工学1本に絞りました．でも，セオリー的なクラスはやっぱり苦手でしたね，目に見えて形で作るものに憧れていました．幸い，Human Computer Interaction（HCI）という，人間とコンピュータのインタラクションを専門に勉強する学科を，著名なTerry Winograd教授が立ち上げた頃でした．また，ユーザーインターフェース（以下UI）は，大学院で扱う課題でしたが，特別に学上用のクラスが作られ，それを受けることができて，そこでUIと本格的に出会いました．

トニー HCIはどんなクラスでしたか？

ブラッド クラスの方式が非常に環境都市学に似ていました．またUIのデザインプロセスも環境都市学に似ていると思いました．たとえば，利用する人を観察したり，何のために何を使うか？など，さまざまな共通点がありました．実際のクラスのほうは，少人数のクラスでおもにディスカッションを行うことが多かったし，プロジェクトもありました．

そこでの初めてのプロジェクトは，ビデオデッキのプログラム機能をデザインするというものでした．古典的なUIの問題です．ビデオデッキの時計合わせができないユーザーって多いですよね．それらを作ったのはエンジニア達なので，使いづらいUIが多かったのが原因だと感じました．クラスでは実際にモックアップを作ったり，テストをしてもらう人の選択やUI設計の基礎を実際のハンズオンで学べたことが良かったです．

次のプロジェクトはアップルコンピュータの図書・資料室の設計でした．図書室の職員はただデータベースの作り変えを望んでいたようなのですが，われわれが調べてみると，社員の多くは図書室の存在すら知らない人が多く，そこから設計をし直すという方向でプロジェクトを進めました．データベースのUIの作り変えももちろん行いましたが，アップルコンピュータ社内で図書・資料室を広く使ってもらうためにアクセスをよくしたりもしました．HCIと出会うことによってUIの仕事が自分に向いていると感じ，自分のキャリアとして進めていこうと決めたのです．

☆ オラクルを経てアップルコンピュータへ

トニー そして，卒業後は？

ブラッド UI設計は，新卒のポジションではないという業界の慣習にまずぶつかりました．ソフトウェアのエンジニアとして経験を積んだり，ある程度の製品開発のプロセスを経験しないと良いUI設計はできないという考え方ですね．それで，とりあえずオラクルに入社しました．当時，ちょうど完全にテキストベースのSQLからGUIが出てくる頃でした．私はMac関連のAPIや社内のツールキットへのポーティングの作業をしました．オーソドックスなソフトウェアエンジニアの仕事でした．その後，Macがビジネス系プラットホームとしての存在価値が薄れていくのを感じて次のことを考えました．

ブラッド・ホックバーグ氏

トニー 新卒でオラクルに行かれたのですね．これもまた意外ですね，ストレートにアップルに行かれると思ってました．

ブラッド 結果的にはアップルに行くのですが，プログラミングの仕事としては，オラクルは良かったです．転職先は，幸いにアップルのPowerTalkのグループで，User Experience Engineerというポジションに就きました．Gil Amelio氏が社長だった頃で，けっこうドロドロしていた時期ではないでしょうか？　のちにMacOSグループに配属されて，いろいろな仕事をしました．ちょうどCoplandが開発されている頃でした．アップルのまったく新しいOSの試みでしたが，6か月ごとに大幅な書き換えをやっていたような気がします．そのたびにこれまでやって来たことをスクラップにしていて，たいへんな作業になっていました．たとえばファイルシステムを全部変えてしまうとかね……．そうするとほとんどどすべてに影響を及ぼしていましたね．出荷がドンドン遅れていたし，既存のOS 8やOS 9にはまったくエンジニアがまわらない状態でした．結局私は，Coplandのグループを離れて，人数の少ないOS 9のプロジェクトに行きました．プロジェクトマネージャ，テクニカルリード，UIデザイン，プログラマとかいくつかの役割を兼務しました．ファインダ，コントロールパネル，インストーラとかユーザーの目に見えるところをかなり担当していました．

トニー 結局Coplandはキャンセルされ，OS XはNeXTの買収で進んだのですよね？

ブラッド そうですね，Ellen Hancock氏が入ってからさまざまな大きな流れが決まり，外部からの買収を視野に入れた戦略に切り替わりました．BeOSがもっとも有力だったのですが，話がまとまらずNeXTに流れたのだと聞いています．

*　　　*

次回の予告

実際のユーザーインターフェースの設計の仕事について，具体的なプロセスや製品についてお話を伺う．

トニー・チン　htchin@attglobal.net　WinHawk Consulting

HARD WARE

●4ビットマイコン

S1C63654

- ROM，RAMやLCDドライバなどを搭載し，使用時の平均消費電力が1μA以下の低消費電力4ビットマイコン.
- 電波時計をはじめサーモメータ，リモコン，温度湿度TAGなどLCD表示付きバッテリ駆動の小型携帯機器に適する.
- 抵抗周波数変換型A-Dコンバータを2チャネル内蔵.
- 抵抗とサーミスタあるいは温度センサを外付けするだけで，一般のA-Dコンバータと比較して1/100～1/1000以下の低消費電力で温度，湿度の測定が可能.
- 電源電圧検出回路を搭載.
- 32×6/5/4/3のLCDドライバを内蔵.
- 時計用8ビットタイマ，1/1000sストップウォッチタイマ，PWM出力が可能な16/8ビットプログラマブルタイマなどの各種タイマ回路を内蔵.

■セイコーエプソン（株）
サンプル価格：¥170
TEL：042-587-5816
URL：http://www.epsondevice.com/

●64Mビットフラッシュメモリ

M29DW640D

- 携帯電話ハンドセット，個人情報デバイス，携帯用パソコン，GPSレシーバなどに適する，業界標準の3VフラッシュメモリM29ファミリを拡充し，最新の0.15μmプロセスを採用した64Mビットデバイス.
- 8または16ビットワードのI/Oオペレーション構成が可能で，四つのバンクA/B/C/D（8/24/24/8Mビット）を装備．二つの8Mビットバンクには，パラメータブロックをそれぞれ八つ装備.
- 25ns/30nsページリードアクセスタイム，12V高速プログラミング（オプション），5つの高速プログラムコマンド，セキュリティ情報などを格納するための256バイト拡張メモリブロックなどの機能を装備.

■STマイクロエレクトロニクス（株）
サンプル価格：¥800
TEL：03-5783-8240　FAX：03-5783-8216

●シンクロナスDRAM

IP00C610

- シンクロナスDRAMを用いて，ディジタル映像，画像処理に用いられる3ポート画像メモリを構成するためのフレームメモリコントローラ.
- 24ビット幅のCPUインターフェース，画像入力および出力ポートを備える.
- 垂直同期信号と水平同期信号を用いて，画像データおよび画像格納用メモリアドレスを2次元で管理して書き込み，読み出しを制御.
- 各画像ポートは非同期に独立して動作し，ノーウェイトで最高80M画素/秒まで動作可能.
- 1チップでモノクロ画像，RGB画像に対応し，専用フィールドメモリやVRAMを使う場合と比較して，大容量の画像メモリをフレキシブルに低コストで実現.

■アイチップス・テクノロジー（株）
価格：下記へ問い合わせ
TEL：06-6492-7277　FAX：06-6492-7388

●USBインターフェースデバイス

ISP1582/83 ISP1183

- ポータブル機器を対象に，省電力かつ高速でローピンカウントを実現.
- 携帯電話，ディジタルカメラ，ディジタルビデオ，PDA，MP3プレーヤなどのコンシューマ製品間をシームレスに接続.
- ISP1582/83インターフェースデバイスは，特許を取得した独自の電力供給回路によって，過剰な電力消費を抑えつつ，高速USB周辺機器の接続が可能．ISP1582はUSBケーブルと搭載システムの双方から電源供給を受けるため，消費電力は45mAと低く，システム電源からは4mA以下の電力しか必要としない.
- ISP1183 USBインターフェースデバイスは，省電力のLPCソリューションを提供すると同時に，最速で12Mbpsの高速データ転送を実現．機器の電源およびその入出力の論理しきい値は1.8Vで設計．パッケージのピンアウトは5×5mmの32ピンで，8ビットの汎用パラレルインターフェースが付属.

■ロイヤル フィリップス エレクトロニクス
価格：下記へ問い合わせ
URL：http://semicon.philips.co.jp/

●無線システムオンチップ

WirelessUSB LS

- 通信距離は最大10メートル，平均レイテンシは4ms秒未満の2.4GHz無線SoC.
- 高度集積無線送信機とディジタルベースバンドを特徴とするテクノロジによって，キーボードやマウス，ビデオゲームコントローラなどのコードレスヒューマンインターフェースなどに適する.
- 2.4GHzグローバルISMバンドで動作するため，地域的な周波数の要求事項に関わらず，世界中にソリューションを展開することが可能.
- 独自のDSSSコード体系を使用して，802.11およびBluetoothネットワークが存在する場合の堅調な動作を確保することにより，近接する数千ものデバイスに対応が可能.
- 62.5kbpsで双方向性または単向性RF伝送を行う，トランシーバまたは送信専用デバイスとして販売される.
- 同社のenCoReのようなUSBコントローラとともに使用することにより，デバイスドライバの開発が不要となる.

■日本サイプレス（株）
価格：¥235～（100,000個時）
TEL：03-5371-1921　FAX：03-5371-1955

●PLL周波数シンセサイザLSI

MB15F72UV，MB15F73UV，MB15F74UV，MB15F76UV

- 低消費電力を実現したULシリーズの後継製品で，BiCMOSプロセスU-ESBIC4と回路設計により，性能を維持したまま汎用品としては最小パッケージ（2.4×2.7×0.45mm）を実現.
- 50MHz～6.0GHzまで，動作周波数別に4製品を用意し，携帯電話やPDA，GPS，無線LANなど，移動体通信のさまざまなアプリケーションに対応.
- 小型BCC-18ピンパッケージに実装.

■富士通（株）
価格：¥140
TEL：03-5322-3390
E-mail：edevice@fujitsu.com

●モバイルFCRAM

MB82DBR08163

- マルチメディア機能を搭載した，第三世代携帯端末の高度なアプリケーションの実現に適する.
- 独自開発の高速，低消費電力型の次世代メモリコアに，SRAMインターフェースを搭載した疑似SRAM.
- 128Mビットのメモリサイズをもち，携帯機器用のメモリに対する大容量化の要求に対応可能.
- バーストモードによる連続読み出し，および連続書き込み動作が可能.
- 66MHz動作時で，バーストモード時のクロックアクセスタイムは12ns.
- 外部クロックに非同期の，ページモードでの読み出し動作を可能にし，ページアクセスタイムは最大で20ns.
- 提供は，71ピンFBGAパッケージに加え，実装用途向けにチップまたはウエハの形態でも可能.
- 最大で200μAのスタンバイ電流を実現.

■ 富士通(株)
価格：¥2,000
TEL：042-532-1416
E-mail：edevice@fujitsu.com

●16ビットD-Aコンバータ

AD9726

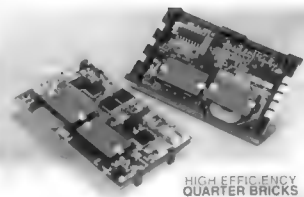
- サンプリングレートが600Mspsを超える，16ビットD-Aコンバータ.
- テストおよび測定機器，計装機器，レーダおよび衛星通信システムなどに適する.
- 出力周波数が100～300MHzで－161dBm/Hz，20MHz出力では－169dBm/Hzと，高いノイズ性能が特徴.
- 相互変調歪み(IMD)を低減することでノイズと帯域幅の問題を解決し，高性能な信号処理と高速な情報処理が可能.
- LVDSレシーバはSDRまたはDDRモードをサポートし，フレキシブルなタイミングインターフェースを搭載.
- 電流調整範囲がフルスケールで2mA～20mAと広く，パワーレベルを下げた動作が可能.
- 電流出力はシングルエンドや差動など，さまざまな回路構成にコンフィグレーション可能.

■ アナログ・デバイセズ(株)
サンプル価格：$35.00(1,000個時)
TEL：03-5402-8128

●DC-DCコンバータ

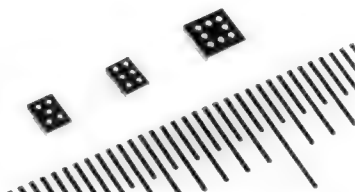
ULQシリーズ

- 業界標準のクォータブリック型で，高効率，単出力絶縁型DC-DCコンバータ.
- 鉛フリーに対応したオープンフレーム形状をし，表面実装モデルとピン実装モデルの2種類を用意.
- 出力電圧，出力電流容量，入力電圧範囲などの特性によって，数十種類のモデルをラインナップ.
- ヒートシールドと呼ばれるケースを本体に被せることで，本体内部の温度上昇を抑える方法を採用.
- 25Aまでの出力電流容量をもつ.
- 1.2，1.5，1.8，2.5，3.3Vの出力電圧モデルを用意.

■ デイテル(株)
価格：¥7,100～(1～24個時)
TEL：03-3779-1031　FAX：03-3779-1030

●EEPROM

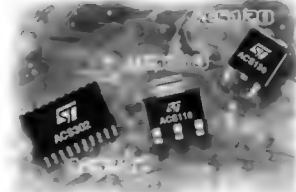
S-24C04/CS16/CS64

- 小型WLPパッケージを採用した，世界最小クラスの2ワイヤシリアルEEPROM(2mm角以下，厚さ0.6mm以下).
- 小型携帯機器をはじめ，Bluetoothモジュールや CCDカメラモジュールなど高密度実装を要求されるアプリケーションに適する.
- S-24C04は4kビット，S-24CS16は16kビット，S-24CS64は64kビット品.
- ボルテージディテクタを内蔵し，電源変動や電源ON/OFF時の誤動作を防止する低電圧時の書き込み禁止機能を装備することにより，機能面での信頼性を向上.
- 小型なため，各種コントローラやDSPなどのデータ格納などの用途に使用できる.

■ セイコーインスツルメンツ(株)
サンプル価格：¥100
TEL：043-211-1193

●ACラインスイッチファミリ

ACS302-5T3/ACS110-7S/ACS120-7S

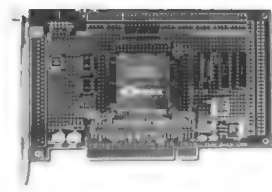
- ACS110-7SおよびACS120-7Sはシングルスイッチバージョン，ACS302-5T3は3スイッチアレイ形式のACラインスイッチ.
- 独自のASDプロセス技術を採用し，家電製品のニーズやEMC標準のIEC61000-4-5規格案件に対応する高電圧保護回路を内蔵.
- 高いゲート感度をもつため，マイクロコントローラからの直接駆動が可能で，低消費電力を実現.
- 最大500Vのブロッキング電圧と最大1100Vのクランピング電圧により，240Vの主電圧からの誘導負荷により電磁的に誘起される跳ね返り電圧に対する耐性を備える.

■ STマイクロエレクトロニクス(株)
サンプル価格：¥200(ACS302-5T3)
¥80(ACS110-7S)
¥90(ACS120-7S)
TEL：03-5783-8240　FAX：03-5783-8216

●リアルタイムシミュレータ

RTSim

- MATLAB/Simulinkモデルを搭載した大規模FPGAボード，32ビット浮動小数点演算IPおよび高速入出力ボードを制御することにより，0.1μs～10μsまでのシミュレーション周期のモデルに対応が可能.
- 32ビット浮動小数点演算IPは，アルテラ社の最新デバイス「Stratix」を採用.
- 高速入出力ボード(A-D，D-A，DI，DO)は，大規模FPGAボードのオプション.
- シミュレーションモデルの設計速度を，従来に比べて20～50倍高速化した.
- シミュレーションモデルをベース部と高速部に分割し，ベース部をCPUによりソフトウェアでモデル計算を行う.

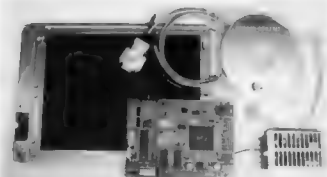
■ ディエスピーテクノロジ(株)
価格：下記へ問い合わせ
TEL：0533-73-1388　FAX：0533-73-1389

■**弊誌では新製品に関するニュースリリースを募集しております.**
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　(編集部)

HARD WARE

●VGA カラー液晶表示学習スタータキット

KS6448-STK7727

- SH7727 に内蔵された LCD コントローラの利用方法や SDRAM の設定を短時間で学習できるスタータキット.
- CPU ボードを添付.
- 10.4 インチ TFT カラー液晶，タッチパネル，タッチパネルコントローラを装備.
- VGA の TFT カラー液晶を SH マイコンで直接コントロール可能.
- C 言語の知識があれば，すぐに利用できる.
- 全回路図添付，量産時のロイヤリティは不要.
- 参考ソフト，マニュアル，ケーブルなどを標準添付.

■（有）ケニックシステム
価格：下記へ問い合わせ
（¥43,800 CPU-501 のみ）
TEL：086-209-0677　FAX：086-209-0673
E-mail：sales@kenic.co.jp
URL：http://www.kenic.co.jp/

●マルチプログラマブル表示器

ACTIVE TOUCH

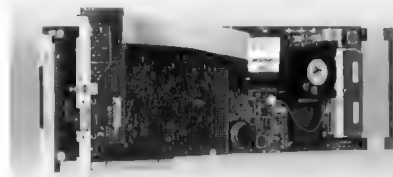
- Web ブラウザや動画，音声などのマルチメディアデータの再生も可能な，マルチファンクション/マルチユースプログラマブル表示器.
- プログラマブル表示器の基本機能である，スイッチランプなどの作画設計を開発マシンで行えるエディタソフトを標準添付.
- ターゲットマシンである本体にも同機能のセルフエディタを内蔵しており，設置時やその後の調整，修正作業を現場で行うことが可能.
- 省配線 I/O によるリモート信号入出力機能に対応するほか，同時に複数種類/メーカーの PLC を接続することが可能.
- 条件判断/繰り返し処理/アラームなどの条件制御を設定可能な，ステップ制御機能を装備.
- OPC サーバ/クライアント技術により，SCADA ソフトやプロセス制御装置などのネットワークノードとの通信や，ホストコンピュータとのターミナル通信による大規模な連携システムにも対応できる.

■（株）コンテック
価格：¥260,000（液晶表示部：10.4 インチ）
¥270,000（液晶表示部：12.1 インチ）
TEL：03-5628-9286　FAX：03-5628-9344
E-mail：tsc@contec.co.jp

●計測制御用アドオンボード

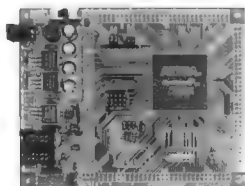
NI PCI-7041/6040E

- 「LabVIEW 7 Express」が提供するリアルタイム計測，テスト，制御アプリケーションをデスクトップ PC に組み込むための 700MHz Pentium III オンボードプロセッサ搭載高速プラグインハードウェア.
- 12 ビットの分解能をもち，最大 250ksps のサンプリングレートの 16 のアナログ入力を搭載.
- 八つのディジタル I/O ライン，二つの 24 ビットのカウンタ/タイマをもち，二つの 12 ビットアナログ出力を搭載.
- ユーザー定義プログラムのための 32M バイト DRAM，32M バイトのコンパクトフラッシュを装備.

■日本ナショナルインスツルメンツ（株）
価格：¥432,000
TEL：03-5472-2970　FAX：03-5472-2977
E-mail：prjapan@ni.com

●Cyclone ブレッドボード

Cyclone ブレッドボード CSP-025/CSP-024 シリーズ

- アルテラ社の高性能，低価格 FPGA を実装した評価用基板の完成品.
- V_{CC}A を除くすべての全ピンをランドに引き出してある.
- 電源回路，リセット回路，クロック源，ISP 可能コンフィグレーション ROM を実装.
- JTAG 用コネクタは，BL3，BL2，BLKIT，ByteBlasterMV，ByteBlaster II などのダウンロードケーブルに対応.
- クロック源は，80/66/40/33/20/16.5MHz のいずれかからジャンパで選択可能.
- コンフィグレーション ROM は，1 万回以上の書き換えが可能な EPCS1 または EPCS4.

■（有）ヒューマンデータ
価格：下記へ問い合わせ
TEL：072-620-2002　FAX：072-620-2003
E-mail：sales2@hdl.co.jp
URL：http://www.hdl.co.jp/

●無線 LAN アクセスポイント

AP-5100

- 54Mbps（5.2GHz/2.4GHz）のデュアル無線 LAN（2 波同時通信）を実現.
- 送信出力可変機能，負荷分散機能により通信の干渉を制御し，小セル化を実現.
- スパニングツリー機能により，通信経路の障害を回避する高信頼なネットワークを構築.
- 802.11b の通信の干渉による，802.11g の通信速度の低下を防ぐ 11g 保護機能をサポート.
- 高速ルーティング機能を搭載し，光ファイバ/ADSL/CATV 回線に対応.
- アクセスポイント間無線通信機能を搭載.
- 暗号化セキュリティ OCB AES，WEP を搭載し，IEEE802.1x/EAP（別途 RADIUS サーバが必要）に対応.

■アイコム（株）
価格：オープン価格
TEL：06-6792-4949

●オンボードデバッグユニット

AM1 Debug Probe Series for MN101CF49K

- 松下電器産業社製マイコン，AM1 シリーズ用オンボードデバッグユニット.
- ターゲットとの接続は PORT15 とシリアルポート，センス用電源，GND を接続するだけでデバッグ環境の構築が可能.
- 電源は USB パワーで動作するため，別途電源は不要.
- ターゲット電源をセンスしているため，5.0V 以下の電圧にも対応.
- ブレークに MN101CF49K の ROM コレクション機能を利用.
- スイッチ設定で，ユーザープログラムをパワーオンスタートさせることが可能.
- Windows 版 C ソースコードデバッガを搭載.

■（株）オブジェクト
価格：¥50,000
TEL：06-6844-1747　FAX：06-6844-1760
E-mail：info@object.co.jp

●Nios開発キット

Stratixプロフェッショナル・エディション

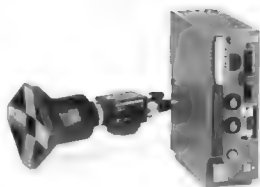
- 最新のNiosエンベデッドプロセッサバージョン3.02，4万個以上のロジックエレメント，および3Mビット以上のオンチップメモリを備えたStratixEP1S40デバイスが含まれる．
- 開発ボードは，16MバイトのSDRAM，1MバイトのSRAM，8Mバイトのフラッシュメモリ，10/100Base Ethernetポート，2個のシリアルポート，ソフトウェアトレースデバッグ用Mictorコネクタ，2個の拡張ヘッダ，電源およびByteBlaster IIダウンロードケーブルを備える．
- Niosプロセッサーバージョン3.02には，拡張OCIコアとファーストシリコンソリューション社から提供されるリアルタイムソフトウェアデバッグ用ソフトウェアが含まれている．
- デザイナにSOPC Builderシステムデザインツールを含む，Quartus IIデザインソフトウェアと完全なソフトウェア開発ツール群を提供．

■日本アルテラ（株）
価格：$2,495
TEL：03-3340-9480　FAX：03-3340-9487
E-mail：japan@altera.com
URL：http://www.altera.co.jp/

●2次元コードリーダ

形V530-R2000

- ウエハ上の2次元コード読み取りに適する固定型．
- コントローラ（本体）とカメラ，専用リーディングヘッド（照明）で構成され，コンソール，ケーブル，モニタなどの周辺機器は従来品のものをそのまま利用可能．
- 読み取り対象ワークの状態や周囲環境が変化した場合でも，最良の撮像状態で読み取れるよう，照明を自動制御．
- アプリケーションに応じて，斜光/同軸照明，リング照明，V型同軸照明の3種類のリーディングヘッドを用意し，設置工数の大幅削減を実現．

■オムロン（株）
価格：オープン価格
TEL：075-344-7069

●IEEE802.1x認証サーバユニット

SVR-RDS（FIT）シリーズ

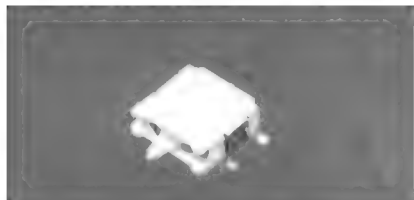
- 無線LANのセキュリティ強化を可能にする専用機．
- 「SVR-RDS（FIT）L」は，小型軽量（80×115×25mm）なコンパクトモデル．
- 「SVR-RDS（FIT）」は，壁掛けや，同社HUB，FLEXLAN DS540シリーズアクセスポイントとのスタックなど，状況に応じたさまざまな設置が可能．
- IEEE802.1xの認証サーバとして機能．
- IEEE802.1xの認証サーバに必要なCA（認証局）とRADIUSサーバ機能を装備．
- 設定や管理は，Webブラウザから可能．

■（株）コンテック
価格：¥148,000〔SVR-RDS（FIT）L〕
¥158,000〔SVR-RDS（FIT）〕
TEL：03-5628-9286　FAX：03-5628-9344
E-mail：tsc@contec.co.jp

●2方向検知用スイッチ

FT-2Wayスイッチ

- 携帯電話，ディジタルカメラ，ディジタルビデオ，PDA，DVDプレーヤ，ノートパソコンなどの小型化が要求される機器に適する．
- 一つのスイッチに1a接点を2個内蔵しているため，左右2方向の検知が可能．
- 5.0×5.0×1.4mmと小型，薄型サイズのため，実装スペースの削減が可能．
- ツイン金メッキ接点の採用により，5μA/DC5Vの微小負荷に対応でき，接触信頼性に優れる．
- モバイル機器の耐環境性能（耐腐食性，耐湿性）が向上し，採用機器の性能向上に貢献．
- 使用周囲温度は，－25～＋80℃．

■松下電工（株）
価格：オープン価格
TEL：06-6908-1131

●Linuxサーバ

Net-Station

- 配線を接続し，電源を入れて10分程度で稼動させることが可能なサーバ．
- 同社オリジナルのCD-ROMから起動可能な「SAVANT-Linux」を採用しているため，ハードディスクで動作するOSに比べて高速な動作を保証．OSはオンメモリで動作する。
- ファイアウォールはもちろん，パケット攻撃が通用しないLinux系のOSを採用し，さらにOS自体はハードディスクではなくCD-ROM上にあるため外部からの書き換えが不可能など，高セキュリティを実現．
- 面倒な設定はSEが行ってから出荷するため不要．
- システムに障害が発生した場合でも，電源を切りリスタートをかければ現状を復帰できる．
- システムのアップグレードもCD-ROMを交換して起動するだけで完了．
- Webサーバ，メールサーバ，ファイルサーバ，プリンタサーバなどに対応．
- クライアントOSはWindows，Macintoshを問わない設計となっている．

■（株）サバン
価格：¥79,000～
TEL：0120-37-6138　FAX：0120-37-7138
URL：http://savant.jp/

●ライセンスサポート

SupportDesk Product Interstage ライセンスセット

- Webシステムの安定稼動を支える基盤ミドルウェア「Interstage」を，IAサーバ「PRIMERGY Windows 2000 Serverインストールタイプ」に標準添付．
- 従来と比較して，初期導入費用を47%削減するだけではなく，トータルコスト面でも16%の削減が可能．
- 短期間でのWebシステム構築が可能となり，「Interstage」の安定稼動機能に加え，操作方法の質問からトラブル解決支援，修正プログラムの提供など，同社の専門サポート体制によりシステムの安心運用を実現．
- Java環境，.NET Framework環境での利用が可能．

■富士通（株）
価格：¥9,400～/月
TEL：03-6252-2174
E-mail：pr@fujitsu.com

■弊誌では新製品に関するニュースリリースを募集しております．
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

SOFT WARE

●GNUディストリビューション

GNUWing

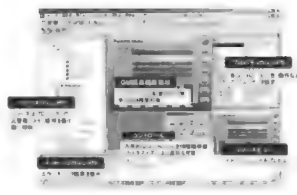
- 複数のプロセッサに対応した最新バージョンのGNUソフトウェアを無償提供.
- GCCの最新バージョンをサポート.
- 日本語マニュアルを用意.
- 操作メニューを日本語化した.
- GNUバイナリユーティリティ「binutils-2.14」, GNUコンパイラコレクション「gcc-3.3」, GNUデバッガ「gdb-5.3」, ロイヤリティフリーライブラリ「newlib-1.11.0」で構成.
- FTP版, パッケージ版, 有償サポート版の3種類を用意.
- 対象ホストは, Windows 2000(SP5以上), Windows XP, Red Hat Linux 9.

■ **アップウィンドテクノロジー・インコーポレイテッド**
価格:無償(FTP版)
¥98,000(パッケージ版)
¥1,200,000～(有償サポート版)
TEL : 045-911-3335　FAX : 045-911-3335

●GUI統合開発環境

GENWARE 2

- ディジタル家電, モバイル端末などのGUI開発を支援する統合開発環境.
- 簡単操作で短期間にGUIが作成可能なGUIエディタを装備.
- C/C++の2種類で, GUIソースコードの自動生成が可能.
- 作成したGUIは, 開発ターゲット試作機のハードウェア完成前に, パソコン上でシミュレーションやデバッグをすることが可能.
- CPUパワーやメモリ量が小さい環境でも十分な性能を発揮する, コンパクトかつ高速なGUIライブラリを実装.

■ **(株)アイ・エル・シー**
価格:下記へ問い合わせ
TEL : 082-262-7799　FAX : 082-263-4411
E-mail : fa@ilc.co.jp
URL : http://www.ilc.co.jp/

●自動車用アプリケーション開発プラットホーム

OSEK/VDX OS

- 「OSEKturbo OS Version 2.2」は, OSEK OSにスケジュール機能を追加し, 自動車用アプリケーションの開発に安定したプラットホームを提供.
- メモリ使用量を改善し, 開発サイクルを短縮するのに役立つように設計.
- システムタイミングの問題を開発の初期段階から明らかにでき, 高信頼性でより安全なソリューションの開発が可能.
- CodeWarrior統合開発環境を利用して, ボードの立ち上げとアプリケーションのタスクを行うことが可能で, プロセッサの初期設定のサポート, プロジェクトのセットアップの簡略化, 異なる半導体アーキテクチャに対しても一貫したルック&フィールを提供することで, 生産性の向上を実現.
- 8, 16および32ビットを含む幅広い範囲のマイクロコントローラをサポート.

■ **メトロワークス(株)**
価格:下記へ問い合わせ
TEL : 03-3780-6091　FAX : 03-3780-6092
E-mail : j-info@metrowerks.com
URL : http://www.metrowerks.co.jp/

●ARM開発ツール

IAR Embedded Workbench for ARM 3.40

- IARシステムズ社が開発した, ARM用組み込みアプリケーション統合開発環境.
- プロジェクトマネージャにより, 関連したすべてのプロジェクトを同一のワークスペースに登録でき, コンパイラおよびアセンブラリストファイルへのアクセスが容易.
- ThreadX OS用のプラグインモジュールをIAR C-SPYデバッガにてサポート.
- ARM VFPフローティングポイントコプロセッサをサポートすることにより, コンパイラ, アセンブラ, デバッガの一連のツールチェインがVFPを認識.
- マクレガー社製のmpDemon JTAGインターフェースをサポート.
- EPI社製Jeeni JTAGエミュレータをシリアル接続に加えて, Ethernet接続でもサポート.
- 各社の評価ボード用サンプルプロジェクトファイルをサポート.

■ **(株)プロトン　ソフトボード事業部**
価格:下記へ問い合わせ
TEL : 03-5337-6431　FAX : 03-5337-6130

●シグナルインテグリティ解析ツール

HyperLynx 7.0

- プリおよびポストレイアウトのシグナルインテグリティ解析と検証を行うツール.
- 500MHz以下のクロック周波数の主流設計用の「HyperLynx EXT」およびマルチギガビット設計用の「HyperLynx GHz」の二つのバージョンを用意.
- ディジタル信号の品質を落とす要因となる, 高速ディジタル基板上で増大するICのクロック周波数の問題に対応.
- 主要なPCBレイアウト環境と互換性があり, 設計の初期段階でSI, クロストークやEMCエラーを予測, 排除することが可能で, 費用がかかるレイアウトやプロトタイプ, テストのサイクルを削減.
- 「HyperLynx EXT」は, インピーダンスプランニングテクノロジやスプレッドシートベースのスタックアップエディタの追加, ディファレンシャル信号の機能拡張やIBISのエンハンスメントなどの機能を多数装備.

■ **メンター・グラフィックス・ジャパン(株)**
価格:下記へ問い合わせ
TEL : 03-5488-3035　FAX : 03-5488-3032

●V_R4131用組み込みOS開発ソフト

NetBSD用 BootLoader4131

- 64ビットMIPS-CPU V_R4131(NEC製)を搭載した組み込みシステムであるタンバック社製MBaseに対応した「boot Loader」「NetBSDバイナリ・差分」「NetBSD開発環境の作成」で構成されるソフトウェアツール.
- MBase+USB Flash disk(128Mバイト)構成で, シャットダウン回避を含めたNetBSDの評価ができ, 独自の機能追加も可能.
- 「boot Loader」のソースコードを参照することにより, MBase以外のV_R41系組み込みシステムへのNetBSDの移植工期を低減.
- 「NetBSD開発環境の作成」の参照により, クロス開発環境構築時間の低減が可能.
- デベロッパーズ版ではV_R41系組み込みシステム開発および製品搭載, スタンダード版ではMBaseでの開発, サンプル版はMBaseでのboot load機能確認がそれぞれ可能.

■ **中央システム技研(株)**
価格:¥500,000(デベロッパーズ版)
¥20,000(スタンダード版)
TEL : 042-321-5441
E-mail : eigyou@gw1.csr.co.jp

●Windowsデバイスドライバ開発支援ツール

DriverStudio 3.0（英語版）

- Visual Studio .NETのIDEおよびVisual Studio 6.0に統合可能で，デバイスドライバ開発の期間短縮，効率化を実現．
- Windows Hardware Quality Labsドライバ認証テストの品質基準を満たすドライバの生成が可能．
- 「DriverWorks」「DriverNetworks」「BoundsChecker Driver Edition」「TrueTime Driver Edition」および「TrueCoverage Driver Edition」のツール群で構成．
- DriverWorkbenchの機能が向上し，「Visual SoftICE」とDriverStudioの各機能との統合を実現．
- 「Visual SoftICE」は，マルチウィンドウ，カスタマイズ可能なGUIをもつ2マシンデバッガ．32ビットのホストマシンで複数の32ビットまたは64ビットのターゲットマシンのデバッグが可能．
- 日本国内で現行バージョンを使っているユーザーは，無償でダウンロードできる．

■日本コンピュウェア（株）
価格：下記へ問い合わせ（無償ダウンロード）
TEL：03-5473-4530　FAX：03-5473-4528
E-mail：marketingjapan@compuware.com
URL：http://www.compuware.co.jp/

●データ放送用ブラウザ

NetFront v3.0 DTV Profile

- 地上ディジタル放送のデータ放送に対応したディジタルテレビ用ブラウザ．
- BML（Broadcast Markup Language）などのARIBで策定しているBS/CSおよび地上波ディジタル放送のデータ放送規格に対応．
- 組み込み用ブラウザの設計思想を継承し，さまざまなプラットホームへの搭載を実現．
- HTMLコンテンツのブラウジング機能については，「NetFront v3.0 DTV Profile Advanced Edition」で対応．
- 「NetFront v3.0 DTV Profile Advanced Edition」では，ディジタルテレビ向けに最適化されたユーザーインターフェースの提供や，Flash 6.0やHelix DNA Clientなどさまざまなプラグインのサブライセンスと追加実装が可能．
- 双方向通信機能や地域限定サービス，事業者ごとのデータ保存などにも対応．

■（株）ACCESS
価格：下記へ問い合わせ
TEL：03-5259-3685　FAX：03-5259-3684
E-mail：prinfo@access.co.jp

●Java開発ツール

Borland Together Edition for Eclipse 6.1 日本語版
Borland Together Edition for WebSphere Studio 6.1 日本語版

- それぞれEclipseとWebSphere Studioの開発環境にシームレスに統合されており，開発者はモデリング，設計，品質保証機能を利用して，高品質なエンタープライズアプリケーションの迅速な構築が可能．
- モデル図，Javaソースコードのいずれかに変更を加えた場合でも，常に同期がとられた状態で維持され，既存のプロジェクトに対してもクラスモデルを自動生成．
- 業界標準のJ2EEパターンやUIパターン，テストケースなどのデザインパターンやテンプレートが搭載されており，テンプレートエキスパートを使用すれば，標準テンプレートのカスタマイズや新規追加も可能．
- 広範囲なフィルタリングは，すべての変更を正しくアプリケーションに反映．
- 品質保証のための総合的な検査，計測機能により，潜在的な問題の早期発見が可能．
- ドキュメント自動生成機能により，HTMLやRTFなど各種ドキュメントの生成が可能．

■ボーランド（株）
価格：¥600,000（指名ユーザーライセンス）
¥950,000（フローティングライセンス）
TEL：03-5350-9380　FAX：03-5350-9369

●汎用ビジュアライゼーションツール

EnSight Ver7.6

- 米国コンピュテイショナル エンジニアリング インターナショナル社が開発した，汎用ビジュアライゼーションツール．
- 可視化するための計算処理を行うサーバプロセスとユーザーインターフェースやグラフィックス処理を実施するクライアントプロセスに分かれた分散処理を採用．
- 流線表示，ベクトル表示，等値面表示，プロット表示などのほかに，立体視やVR機器を用いた可視化処理をリアルかつ高速に実施．
- 非定常データのコントロールを容易にする時系列機能，加速コントロール機能などをもつ．
- 視点位置のリセットもしくは，視点ファイルの読み込みが補間動画作成中でも実行可能となり，最短経路で現在の視点位置と新たな視点位置を接続する．
- 動画作成を簡易化するクイックアニメーション機能により，旋回，回転，分解図表示の設定が可能．

■サイバネットシステム（株）
価格：¥2,100,000～（買い取り）
¥840,000～（レンタル）
TEL：03-5978-5406　FAX：03-5978-5960
E-mail：mcaeinfo@cybernet.co.jp

●プログラム仕様書自動生成ツール

CasePlayer2 Ver.2.1

- ソースファイルを登録するだけで，プログラムロジックを解析し，各種ドキュメントを自動生成．
- 一括HTML変換機能により，ブラウザでの仕様書の閲覧が可能．
- 作成された仕様書を統合化する，仕様書ブラウザを搭載．
- 組み込み向けC言語，各MPUアセンブラに対応．
- C言語プログラム構文解析機能「Source Doctor」を装備．
- ソースの印刷，行番号表示，文字列検索が可能．
- フローチャートやモジュール構造図が，印刷イメージどおりにページ分割されて画像ファイルとして生成され，HTMLファイルでの保存が可能．
- ハイパーリンクにより，各仕様書間のジャンプが可能．

■ガイオ・テクノロジー（株）
価格：下記へ問い合わせ
E-mail：case@gaio.co.jp
URL：http://www.gaio.co.jp/

●組み込みファイルシステム

GR-FILE

- 組み込み機器開発に適する，FATファイルシステム．
- UNIX/Windows，C言語標準I/O互換ライブラリインターフェースを提供．
- 階層化マウントからファイルパス上へ，ファイルシステムの追加が可能．
- メディアに応じた，ファイルシステムごとのキャッシング方式が可能．
- OSに非依存の設計．
- FAT12/16/32に対応し，ロングファイルネームをサポート．
- マルチタスク同時アクセスをサポート．
- ファイルインデックスとデータを分離キャッシング．
- キャッシング，ブロックサイズなどのパラメータ設定の変更が可能．
- ANSI Cで記述されたソースコードでの提供．
- ファイルシステム依存レイヤを分離し，他形式のファイルシステムを追加でサポート．

■（株）グレープシステム
価格：下記へ問い合わせ
TEL：045-222-3761　FAX：045-222-3759
E-mail：info@gr.grape.co.jp
URL：http://www.grape.co.jp/

■**弊誌では新製品に関するニュースリリースを募集しております．**
宛先は，〒170-8461 東京都豊島区巣鴨1-14-2 Interface編集部ニュースリリース係
FAX：(03)5395-2127，E-mail：mngnews@cqpub.co.jp　（編集部）

海外・国内イベント/セミナー情報

INFORMATION

海外イベント

9/30-10/1　**Embedded Systems Conference Asia**
Lakeshore Hotel, Hsinchu, Taiwan
CMP Media LLC.
http://www.english.escasiaexpo.com/

9/30-10/2　**Communications Design Conference**
San Jose Convention Center, San Jose, CA, USA
CMP Media LLC.
http://cmp.iconvention.com/cdc/V40/

10/6-9　**6th Asia-Pacific ITS Forum & Exhibitions**
Taipei International Convention Center, Taipei, Taiwan
Round Table Professional Conference Organizer
http://www.its-taiwan.org.tw/

10/8-12　**KOREA ELECTRONICS SHOW 2003**
COEX, Seoul, Korea
Electronic Industries Association of Korea
http://www.kes.org/kes2003/intro.htm

10/9-13　**Taipei International Electronics Show**
Taipei World Trade Center Exhibition Hall, Taipei, Taiwan
China External Trade Development Council, Taiwan Electrical and Electronic Manufacturer's Association
http://www.taipeitradeshows.com.tw/equipment/

10/13-14　**2003 Optical Storage Symposium**
La Costa Resort and Spa, Carlsbad, CA, USA
OSTA (Optical Storage Technology Association)
http://www.osta.org/oss/

10/13-17　**OptiComm 2003**
OMNI Richardson Hotel, Dallas, TX, USA
SPIE-International Society for Optical Engineering, IEEE Dallas Section, National Science Foundation
http://www.opticomm.org/

国内イベント

10/7-11　**CEATEC JAPAN 2003**
日本コンベンションセンター(幕張メッセ, 千葉県千葉市)
CEATEC JAPAN運営事務局
http://www.ceatec.com/index.html

10/8-10　**日経ナノテクフェア 2003**
東京国際展示場(東京ビッグサイト, 東京都江東区)
日本経済新聞社
http://www.nikkei-nanofair.com/

10/16-17　**組込みソフトウェアシンポジウム 2003**
機械振興会館(東京都港区)
情報処理学会ソフトウェア工学研究会
http://www.seto.nanzan-u.ac.jp/~watanabe/ess03/

10/29-30　**Internet & Mobile 2003**
マイドームおおさか(大阪府大阪市)
日本能率協会
http://www.jma.or.jp/im/

10/29-11/1　**FPD International 2003**
パシフィコ横浜(神奈川県横浜市)
日経BP社
http://expo.nikkeibp.co.jp/lcd/ja/index.html

11/5-7　**計測展 2003 TOKYO**
東京国際展示場(東京ビッグサイト, 東京都江東区)
(社)日本電気計測器工業会
http://expo.nikkeibp.co.jp/jemima/

11/11-13　**Mobile & Wireless World / Tokyo 2003**
東京国際フォーラム(東京都千代田区)
(株)IDGジャパン
http://www.idg.co.jp/expo/mww/

11/12-14　**Embedded Technology 2003 / 組込み総合技術展**
パシフィコ横浜(神奈川県横浜市)
(社)日本システムハウス協会
http://www.jasa.or.jp/et/

開催日, イベント名, 開催地, 問い合わせ先の順

セミナー情報

PC実習!!Javaによるプロセス指向と並列プログラミング入門
開催日時　：9月29日(月)～9月30日(火)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　：68,000円
問い合わせ先：(株)ソフト・リサーチ・センター, ☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_253.htm

オブジェクト指向技術によるプロセス改善の実践
開催日時　：10月2日(木)
開催場所　：CQ出版セミナールーム
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局, ☎(03)5395-2125, FAX(03)5395-1255

Linux GUIプログラミング
開催日時　：10月6日(月)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区)
受講料　：46,000円
問い合わせ先：(株)エイチアイICP事業部, ☎(03)3719-8155, FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/linux/clinuxgui.asp

組み込み型ソフトウェアのための仕様書の書き方
開催日時　：10月10日(金)
開催場所　：オームビル(東京都千代田区)
受講料　：79,800円
問い合わせ先：(株)トリケップス, ☎(03)3294-2547, FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/031010a.htm

オブジェクト指向システム分析/設計[基礎]
開催日時　：10月14日(火)～10月15日(水)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　：76,000円
問い合わせ先：(株)ソフト・リサーチ・センター, ☎(03)5272-6071
http://www.src-j.com/teiki_no/UML/uml_01.htm

制御用コンピュータI/O操作プログラミング
開催日時　：10月15日(水)～10月17日(金)
開催場所　：高度ポリテクセンター(千葉県千葉市)
受講料　：30,000円
問い合わせ先：雇用・能力開発機構 高度ポリテクセンター事業課, ☎(043)296-2582
http://www.apc.ehdo.go.jp/

CANプログラミング基礎コース
開催日時　：10月15日(水)～10月17日(金)
開催場所　：半導体トレーニングセンター 飯田橋会場(東京都新宿区)
受講料　：30,000円
問い合わせ先：(株)ルネサス テクノロジ 半導体トレーニングセンター, ☎(03)3266-9344
http://www.renesas.com/jpn/support/seminar/

JPEG2000, MotionJPEG2000徹底解説
開催日時　：10月16日(木)
開催場所　：CQ出版セミナールーム
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局, ☎(03)5395-2125, FAX(03)5395-1255

Visual Studio .NETによるXML Webサービス構築(C#編)
開催日時　：10月16日(木)～10月17日(金)
開催場所　：NRI大手町ラーニングセンター(東京都千代田区)
受講料　：105,000円
問い合わせ先：(株)エイチアイICP事業部, ☎(03)3719-8155, FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/nrims/ms_xmlwebc.asp

無線LAN IEEE802.11a/b/g　～物理層の規格と測定～
開催日時　：10月20日(月)
開催場所　：アジレント・テクノロジー(株)新横浜オフィス(神奈川県横浜市)
受講料　：42,436円
問い合わせ先：アジレント・テクノロジー(株)計測お客様窓口, ☎0120-421-345
http://jp.tm.agilent.com/tmo/education/course5_03s.shtml

JavaXMLプログラミング入門
開催日時　：10月22日(水)
開催場所　：ディーアイエステクノサービス研修室(東京都文京区)
受講料　：45,000円
問い合わせ先：(株)エイチアイICP事業部, ☎(03)3719-8155, FAX(03)3793-5109
http://icp.hicorp.co.jp/seminar/Java/j_xmlpro.asp

リアルタイムOSの基礎
開催日時　：10月23日(木)
開催場所　：CQ出版セミナールーム
受講料　：13,000円
問い合わせ先：エレクトロニクス・セミナー事務局, ☎(03)5395-2125, FAX(03)5395-1255

USBデバイス開発・設計手法
開催日時　：10月28日(火)～10月29日(水)
開催場所　：オームビル(東京都千代田区)
受講料　：68,500円(1口で1社3名まで受講可)
問い合わせ先：(株)トリケップス, ☎(03)3294-2547, FAX(03)3293-5831
http://www.catnet.ne.jp/triceps/sem/c031028a.htm

ICタグ そのしくみとアプリケーション
開催日時　：10月31日(金)
開催場所　：SRCセミナールーム(東京都高田馬場)
受講料　：48,000円
問い合わせ先：(株)ソフト・リサーチ・センター, ☎(03)5272-6071
http://www.src-j.com/seminar_no/23/23_207.htm

日程はすべて予定です. 問い合わせ先にご確認のうえ, お出かけください.

2003年9月号特集
「C/C++によるハードウェア設計入門」
に関して

▷ SpecCやSystemCなどのC/C++ベースの開発は今後どんどん増えてくようだ．新人などの短期間での戦力化にも有効だといわれている．ハードウェア設計はC++どころかUMLに近づくだろうし，近づいてほしいと考えているが，現場で機械に触れたことのない機械設計者の設計や，現場に出たことのない事務屋のOA化の二の舞にならないでほしい． （伊藤 啓）

興味のあった記事（2003年9月号で実施）

① 序章
② 第1章 SystemCの基礎
③ TOPPERSで学ぶRTOS技術（第1回）
④ 第2章 組み合わせ回路とSystemC記述
⑤ 第3章 順序回路とSystemC記述
⑥ 家電機器をネットワーク化するアーキテクチャUniversal Plug and Playの全貌（第3回）
⑦ 第4章 ステートマシンのSystemC記述
⑧ GNU開発ツール入門
⑨ ITRONのソフトウェアグループ管理
⑩ 組み込みLinuxをとりまく世界（第2回）
⑪ 第6章 SystemC 3.0のロードマップ
⑫ シニアエンジニアの技術草子（参拾壱之段）
⑬ 第5章 SpecCによる協調設計の実際
⑭ InterGiga No.31
⑮ XScaleプロセッサ徹底活用研究（第3回）
⑯ 移り気な情報工学（第34回）
⑰ Engineering Life in Silicon Valley（対談編）
⑱ プログラミングの要（第6回）
⑲ JTAGデバッグツール「WIND POWER *ICE/IDE*」の概要
⑳ SUPERCOMM 2003 Atlanta
㉑ ハッカーの常識的見聞録（第33回）
㉒ RSA Conterence 2003 Japan
㉓ 開発環境探訪（第21回）
㉔ 画像検査アルゴリズムの検証

特集『C/C++によるハードウェア設計入門』についてのアンケートの結果

Q1 C/C++言語ベースのシステム設計に期待することはなんですか？
① HW/SW協調設計（67%）
② 大規模なシステムの開発（17%）
③ システム開発サイクルの短縮（8%）
④ ソフトウェア技術者によるハードウェア設計（8%）
⑤ その他（0%）

Q2 現在どのような手法でハードウェア設計を行っていますか？（複数回答可）
① 回路図（33%）
② VHDL（20%）
③ Verilog-HDL（7%）
④ その他のHDL（13%）
⑤ SystemC（0%）
⑥ SpecC（0%）
⑦ その他のCベース設計言語（0%）
⑧ その他（27%）（ハードウェア設計はしない）

Q3 C/C++言語ベースのシステム設計は主流になると思いますか？
① 1年以内に主流になる（8%）
② 1～3年後には（42%）
③ 3～5年後には（25%）
④ ならない（25%）
⑤ その他（0%）

特集担当デスクから

☆2号連続特集の後編，キャッシュやMMU，割り込み，そして命令セットアーキテクチャについての解説はいかがだったでしょうか．先月号をご覧いただいていない方は，ぜひ先月号も併せてご覧ください．
☆Z80を使っていた人が8086を使い始めたとき，INT 21hによるファンクションコールを「ソフトウェア割り込み」と呼ぶことに違和感を感じたという話を聞きました．いろいろなMPUを実際に使ってみないと，その本質を理解するのは難しいようです．
☆VLIWやJava関連，コンフィギュラブルコンピュータなど，今回触れていない話題も多々あります．今後にご期待ください．

Interface

次号予告 具体例で学ぶ組み込みソフトの再利用技術

2003年12月号は10月25日発売です

組み込み機器と組み込みソフト/オブジェクト指向設計/プロダクトライン分析/電子ポット商品群/UML/障害分析/テスト分析/組み込みシステムのテスト

組み込み機器開発は，日本の「お家芸」といえる．しかし最近の傾向として，世界的に組み込み機器への関心が高まり，これまで組み込み機器の開発を経験したことのない技術者が組み込み機器を開発し，ワールドワイドな市場で製品展開をしようとしている現状がある．組み込みLinuxやCE.netの展開も，その状況を後押ししている．一方，日本の組み込み機器開発現場では，市場からの要求は多様化しながらも，開発サイクルまで短くなるという，つらい状況が続いている．

そこで次号特集では，実際に組み込み機器開発で得た成功体験をもとに，組み込み機器の特徴を分析して再利用技術で問題解決を行うというアプローチを，できるだけ具体的な例を使い，実装例まで含めてていねいに解説する．

★次号には，『シニアエンジニアの技術草子』が別冊付録として付きます！

編集後記

■今日は9月1日．我が家の小1/小3の子供は，期待と不安ではちきれそうな状態で今朝，学校へ向かいました．思えば約30年前，自分もそうだったか．1か月の夏休みから復帰，なんていうパターンは社会人になってからはないけれど，そういうのを経て，今の自分があるのを少し不思議に感じました．がんばれ，子供たち！（洋）

■10月号11月号と2号連続特集の編集が続き，おまけに間にInterface増刊の編集が入るという制作スケジュールで，これまで夏休みが取れませんでしたが，これ(本号)が終わればやっと夏休みが取れるかも？ いや，すぐ後ろに2004年1月号の足音がヒタヒタと……1号休んでまたすぐ特集担当だったりするんです(T_T)（M）

■某社の某誌編集部へ遊びに行ってきました．その雑誌の編集長を交えて小一時間ほど話し込んだのですが，読者層に合わせた発信のしかた，誌面の作り方や方向性など，いろいろと勉強になりました．組み込み方面でもSWESTなどがありますが，同業他社の人との横のつながりは仕事上でも刺激になります．（み）

■休日の朝，自宅のPCを立ち上げてメールチェックすると，440通受信中の表示が！ 内訳はニュースが2通，MLが2通，DMが3通，それ以外の433通はW32.Sobig.F@mmが添付された爆弾メールという悲惨な状況．ウィルスチェッカを導入していたので感染は免れましたが，1日のやる気をすべて削がれました．(@)

■家電にもLinuxが採用され始めて，できる技術者の求人が増えている．しかし，組み込みがわかってLinuxがわかる人など，そんなにいるわけがない．だから今は，できる人は自分を高く売るチャンスだと思う．また，企業側も枠にはまった考えは捨てて，優秀な人材にはもっと投資すべきではないか．（Y）

■新種のコンピュータウイルスが話題になっています．自宅のPCはここ何か月間も放ってある状態なので心配です．メールも携帯で済ませているし，インターネットも使い始めた頃のようには見ていないので，このままでも困らないと言えば困らないのですが，いつまでもこのままにしておくわけにもいかず……．（Y2）

■プロレスとプロ格闘技は別物なのか？ 答えは「NO」であり「YES」であろう．しかし，プロレスラーは強いのか弱いのかと聞かれれば私は瞬時に「YES」と答える．彼らは相手の技を極限まで受けきる肉体的，精神的な「強さ」を持つばかりか対戦相手を光らせる「上手さ」を兼ねそろえているのだ．プロレス頑張れ！(ちゃん)

■先日，某テレビ番組で徳川家康の子孫の方が出演されていました．そして番組の最後に現代の若者に対して，国内に留まらずもっと世界に出て活躍して欲しいというメッセージを残していました．鎖国を実施した将軍の子孫が国外に目を向けようと言う．時代は変わるものですね．因みに今年は江戸開幕400周年．（ふ）

お知らせ

▶読者の広場
本誌に関するご意見・ご希望などを，綴じ込みのハガキでお寄せください．読者の広場への掲載分には粗品を進呈いたします．なお，掲載に際しては表現の一部を変更させていただくことがありますので，あらかじめご了承ください．

▶投稿歓迎
本誌に投稿をご希望の方は，連絡先(自宅/勤務先)を明記のうえ，テーマ，内容の概要をレポート用紙1～2枚にまとめて「Interface投稿係」までご送付ください．メールでお送りいただいても結構です(送り先はsupportinter@cqpub.co.jpまで)．追って採否をお知らせいたします．なお，採用分には小社規定の原稿料をお支払いいたします．

▶本誌掲載記事についてのご注意
本誌掲載記事には著作権があり，示されている技術には工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は，所有者の許諾が必要です．また，掲載された回路，技術，プログラムなどを利用して生じたトラブルについては，小社ならびに著作権者は責任を負いかねますので，ご了承ください．
本誌掲載記事をCQ出版(株)の承諾なしに，書籍，雑誌，Webといった媒体の形態を問わず，転載，複写することを禁じます．

▶コピーサービスのご案内
本誌バックナンバーの掲載記事については，在庫(原則として24か月分)のないものに限りコピーサービスを行っています．コピー体裁は雑誌見開きの，複写機による白黒コピーです．なお，コピーの発送には多少時間がかかる場合があります．

- コピー料金(税込み)
  1ページにつき100円
- 発送手数料(判型に関わらず)
  1～10ページ：100円，11～30ページ：200円，31～50ページ：300円，51～100ページ：400円，101ページ以上：600円
- 送付金額の算出方法
  総ページ数×100円＋発送手数料
- 入金方法
  現金書留か郵便小為替による郵送
- 明記事項
  雑誌名，年月号，記事タイトル，開始ページ，総ページ数
- 宛て先
  〒170-8461 東京都豊島区巣鴨1-14-2
  CQ出版株式会社 コピーサービス係
  (TEL：03-5395-4211，FAX：03-5395-1642)

▶お問い合わせ先のご案内

- 在庫，バックナンバー，年間購読送付先変更に関して
  販売部：03-5395-2141
- 広告に関して
  広告部：03-5395-2133
- 雑誌本文に関して
  編集部：03-5395-2122

記事内容に関するご質問は，返信用封筒を同封して編集部宛てに郵送してくださるようお願いいたします．筆者に回送してお答えいたします．

Interface

©CQ出版(株) 2003 振替 00100-7-10665
2003年11月号 第29巻 第11号(通巻第317号)
2003年11月1日発行(毎月1日発行)
定価は裏表紙に表示してあります

発行人／増田久喜
編集人／相原 洋
編集／大野典宏 村上真紀 山口光樹 落合幸喜 小林由美子
デザイン・DTP／クニメディア株式会社
表紙デザイン／株式会社プランニング・ロケッツ
本文イラスト／森 祐子
広告／澤辺 彰 中元正夫 菅原利江

発行所／CQ出版株式会社 〒170-8461 東京都豊島区巣鴨1-14-2
電話／編集部 (03)5395-2122 URL http://www.cqpub.co.jp/interface/
広告部 (03)5395-2133 インターフェース編集部へのメール
販売部 (03)5395-2141 supportinter@cqpub.co.jp
CQ Publishing Co.,Ltd.／1-14-2 Sugamo, Toshima-ku, Tokyo 170-8461, Japan
印刷／クニメディア株式会社 美和印刷株式会社
製本／星野製本株式会社

日本ABC協会加盟誌
(新聞雑誌部数公査機構)
ISSN0387-9569

Printed in Japan